

---

**User's Guide**

---

**HP B3641 68000 Family  
Cross Assembler/Linker/  
Librarian**

---

## Notice

**Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.** Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

© Copyright 1988, 1990, 1991, 1993, Hewlett-Packard Company.

This document contains proprietary information, which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

IBM is a registered trademark of International Business Machines Corporation.

MS and MS-DOS are registered trademarks of Microsoft Corporation.

Windows is a trademark of Microsoft Corporation.

Microtec is a registered trademark of Microtec Research Inc.

SunOS, SPARCsystem, OpenWindows, and SunView are trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

**Hewlett-Packard Company**  
**P.O. Box 2197**  
**1900 Garden of the Gods Road**  
**Colorado Springs, CO 80901-2197, U.S.A.**

**RESTRICTED RIGHTS LEGEND** Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in subparagraph (C) (1) (ii) of the Rights in Technical Data and Computer Software Clause in DFARS 252.227-7013. Hewlett-Packard Company, 3000 Hanover Street, Palo Alto, CA 94304 U.S.A.

Rights for non-DOD U.S. Government Departments and Agencies are set forth in FAR 52.227-19(c)(1,2).

### **About this edition**

Many product updates and fixes do not require manual changes, and manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual revisions.

Edition dates and the corresponding HP manual part numbers are as follows:

Edition 1	64870-90902, March 1987 E0387
Edition 2	64870-90902, August 1987 E0887
Edition 3	64870-90902, April 1988 E00488
Edition 1	64874-97000, June 1989
Edition 4	64870-97001, December 1989
Edition 2	64874-97002, September 1990
Edition 5	64870-97002, October 1990
Edition 6	64870-97004, October 1991
Edition 3	64874-97004, October 1991
Edition 1	B3641-97000, -97001, -92000, May 1993
<b>Edition 2</b>	<b>B3641-97002, September 1993</b>

### **Certification and Warranty**

Certification and warranty information can be found on the pages before the back cover.

---

## In this Book

If you will be using the assembler with the **cc68k** compiler, you may not need to use this manual, because **cc68k** calls the assembler with the appropriate options.

This book is divided into two parts:

### Quick Start

This part contains:

- installation instructions
- a brief tutorial
- printed copies of the on-line manual pages

### Reference

This part contains detailed reference information about the software, including:

- descriptions of listing formats
- details about assembler, linker, and librarian operation
- descriptions of assembler error messages

This book does *not* discuss how to use assembly language.

---

# Contents

---

## Part 1 Quick Start Guide

### 1 Getting Started

Installing on an HP Workstation 2

Installing on a Sun Workstation 2

Installing on a PC (MS-DOS) 2

Installing on a PC (Windows) 3

Description of the Example Program 4

The "mov\_mesg.s" Program Module 4

The "transfer.s" Program Module 7

The "delay.s" Program Module 8

Assembling the Program Module Source Files 9

Creating an Example Library File 15

Linking the Program Module Relocatable Object Files 17

Linking the Object Modules 18

### 2 Command Syntax

File Extensions 25

as68k(1) 26

ld68k(1) 32

ar68k(1) 44

---

## Part 2 Reference

### 3 Assembler Introduction

as68k Features	52
Assembler Statements	53
Label Field	54
Operation Field	54
Operand Field	54
Comment Field	54
Statement Examples	55
Instruction Statement	55
Directive Statement	56
Macro Statement	56
Comment Statement	56
Return Codes	57
Assembler Syntax	57
Assembler Character Set	58
Symbols	59
Reserved Symbols	60
Location Counter Symbol (*)	61
Symbol Types	61
Constants	62
Integer Constants	62
Floating-Point Constants	63
Character Constants	64
Expressions	66
Assembler Listing Description	68
Assembler Listing	68
Cross Reference Table Format	70

## 4 Instructions and Address Modes

Instructions	73
Qualifiers	74
Scope Qualifiers	74
Floating Point Qualifiers	75
Mnemonics	75
Floating Point Mnemonics	80
Variants of Instruction Types	83
Instruction Operands	84
Registers	84
Address Modes	88
The 68000 Model	90
The 68020 Model	91
The 68332 Model	92
Explanations of Address Modes	93
68881 Floating-Point Coprocessor and Address Modes	98
68040 Floating-Point Unit and Address Modes	99
Assembler Syntax for Effective Address Fields	100
Rules of Assembler Syntax	100
Operand Syntax and Addressing Modes	102
How Code is Generated for Forward Defined Symbols	111
User Control of Address Modes	112
A2-A5 Relative Addressing	114
Address Register Indirect with Displacement Modes	114
Advantages of A2-A5 Relative Addressing	115

## 5 Relocation

Program Sections	125
Common vs. Noncommon Attributes	125
Short vs. Long Attributes	126
Section Alignment Attribute	126
Section Contents Attributes	127
Other Things to Know About Sections	129
How the Assembler Assigns Section Attributes	130
Linking	131
Relocatable vs. Absolute Symbols	132
Relocatable Expressions	133
Label Alignment	135

## 6 Assembler Directives

Notation	141
ALIGN	142
CHIP	143
COMLINE	145
COMMON	146
DC	148
DCB	151
DS	153
ELSEC	155
END	156
ENDC	157
ENDR	158
EQU	159
FAIL	161
FEQU	162
FILE	164
FOPT	165
FORMAT, NOFORMAT	166
IDNT	167
IFEQ, IFNE, IFGT, IFGE, IFLT, IFLE	168
IFC, IFNC	169



IFDEF, IFNDEF	171
INCLUDE	172
[NO]INTFILE	173
IRP	174
IRPC	175
LIST	176
LLEN	177
MASK2	178
NAME	179
NOLIST	180
NOOBJ	181
NOPAGE	182
OFFSET	183
OPT	185
ORG	191
PAGE	193
PLEN	194
REG	195
REPT	196
RESTORE	197
SAVE	198
SECT, SECTION	199
SET	201
SPC	202
TTL	203
XCOM	204
XDEF	205
XREF	206

## 7 Macros

Macro Heading	211
Macro Body	212
Macro Terminator	213
Macro Call	214
LOCAL - Define Local Symbol	217
MEXIT - Alternate Macro Exit	219
Macro Parameter Count	220

## 8 Structured Control Statements

Structured Control Expressions	223
--------------------------------	-----

FOR...ENDF Loop	225
IF ... THEN ... ELSE ... ENDI Conditional Execution	227
REPEAT ... UNTIL Loop	229
WHILE ... ENDW Loop	230
BREAK - Premature Loop Exit	231
NEXT - Proceed to Next Loop Iteration	232
Structured Directive Nesting	233
Structured Directive Listings	234

## **9 Linker/Loader Introduction**

Linker/Loader Features	237
Linker/Loader Operation	237
Program Sections	238
Absolute Section	238
Relocatable Section	238
Noncommon Section	239
Common Section	239
Short Section	239
Long Section	240
Section Alignment	240
Section Contents	240

HP Section Type	241
Memory Space Assignment	241
Incremental Linking	243
Relocation Types	243
Generating HP Format Absolute Files	244
Return Codes	245
Loader Listing Description	245
Loader Listings	245

## 10 Linker/Loader Commands

Summary of Commands	250
Command Format	253
Processing Order	253
;(Comment)	255
# (Continuation)	256
ABSOLUTE	257
ALIAS	259
ALIGN{MOD}	260
BASE	261
[UPPER]CASE, [LOWER]CASE	262
CHIP	264
COMMON	266
CPAGE	267
[NO]DEBUG_SYMBOLS	268
END	269
ERROR, WARN, NOERROR	270
EXIT	271
EXTERN	272
FORMAT	273
INCLUDE	274
INDEX	276
Purpose of the INDEX Command	277
INITDATA	278
[NO]INTFILE	281

LIST	282
LISTABS	285
LISTMAP	286
LOAD	287
LOAD_SYMBOLS	289
MERGE	290
NAME	292
NLIST	293
NOPAGE	295
ORDER/SORDER	296
PAGE	299
PUBLIC	300
RESADD/RESMEM	302
SECT	304
SECTSIZE	305
START	306

## **11 Librarian Introduction**

Librarian Features	309
Librarian Operation	310
Librarian Function -- Overview	310
Command Syntax	316
Use of Special Characters	316
Blanks	317
Command File Comments	317
Module Names	317
Return Codes	318
Library Listing Format	318
Sample Test Program Description	318
Example Librarian Listing	319
Description of Example	320

Brief Format Example Library Listing	320
Brief Format Listing Description	320

## **12 Librarian Commands**

Command Summary	322
ADDLIB	323
ADDMOD	324
CLEAR	325
CREATE	326
DELETE	327
DIRECTORY	328
END, EXIT, QUIT	330
EXTRACT	331
FULLDIR, LIST	332
HELP	333
OPEN	335
REPLACE	336
SAVE	337

### **A Assembler Error Messages**

### **B Loader Error Messages**

### **C Librarian Error Messages**

### **D Error Message Formats**

Error Classes	366
Warnings	366
Errors	366
Fatal Errors	367

Interactive and Non-Interactive Conditions 368

## **E Converting to HP B3641 Assembly Language**

Converting HP 64845 Assembly Language Programs 370

Converting HP 64845 Pseudo-Ops 373

Converting HP 64845 Operands 377

Converting Character Constants 378

Converting Logical Operators 378

Converting HP 64845 Macros 379

Macro Headings 379

Unique Label Generation 379

Conditional Assembly Within Macros 380

Indexing Parameters 381

Converting HP 64845—Miscellaneous 382

White Space 382

Compatibility with older HP 64870 and HP 64874 Files 383

Relocatable and Library Files 383

Assembly Source Files 384

## **F About this Version**

Version 2.01 386

PC Platform Support 386

Re-organized manual 386

Version 2.00 386

Combined products 386

New features: as68k 386

New features: ld68k 388

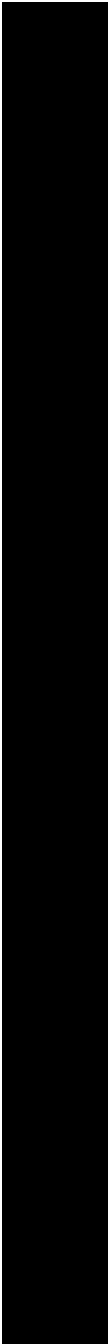
---

# Part 1

---

## Quick Start Guide

Part 1





---

# 1



---

## Getting Started

Installing and using the assembler, linker, and librarian.

---

## Installing on an HP Workstation

This software uses standard HP-UX installation procedures. Look for installation instructions in your *HP-UX System Administration* manual.

---

## Installing on a Sun Workstation

Look for installation instructions in the *Software Installation Guide*, which is packaged with the tape.

---

## Installing on a PC (MS-DOS)

To install from MS-DOS:

- 1 Insert the assembler disk into the floppy disk drive.
- 2 Enter (if the floppy drive is drive A:)

```
a:\install
```

Follow the instructions on the screen.

You will be asked to enter the installation path. The default installation path is C:\hpas68k. The default installation path is shown wherever files are discussed in this manual.

---

## Installing on a PC (Windows)

To install from Microsoft Windows:

- 1 Start MS Windows in the 386 enhanced mode.
- 2 Insert the assembler disk into floppy disk drive A or B.
- 3 Choose the File→Run... (ALT, F, R) command in the Windows Program Manager. Enter "a:\setup" (or "b:\setup" if you installed the floppy disk into drive B) in the Command Line text box.

Then, choose the OK button. Follow the instructions on the screen.

You will be asked to enter the installation path. The default installation path is C:\hpas68k. The default installation path is shown wherever files are discussed in this manual.

**Setup.exe** may not be included with some versions of the assembler. In that case, open a DOS window and use **install.bat**.

## Description of the Example Program

The example programs in this chapter have been included with your 68000 Family Assembler/Linker/Librarian software and can be found in directory:

```
/usr/hp64000/demo/languages/B3641 (UNIX)
```

or

```
\hpa68k\examples (DOS)
```

The examples in this manual assume you are using a UNIX system. If you are using DOS, you may need to adjust some of the path names and file extensions.

The example program moves data from three different memory locations to a fourth memory location. The program uses three modules to show how several program modules are linked together.

The **mov\_mesg.s** program module is made up of a data table which contains the messages to be transferred, the main program which will define a macro and call "transfer" and "delay" subroutines, and a RAM location where the messages will be transferred.

The **transfer.s** program module contains the "transfer" subroutine which is called by the main program. The **transfer.s** subroutine will transfer a message from the data table to the destination memory location. The address of the message to be transferred will be passed in register A0, and the length of the message will be passed in register D0.

The **delay.s** program module contains the "delay" subroutine which is called by the main program. The **delay.s** subroutine will delay for the number of seconds which are passed in register D0.

The **delay.o (delay.obj)** and the **transfer.o (transfer.obj)** relocatable object files will be placed into an example library file called **exlib.a (exlib.lib)**.

### The "mov\_mesg.s" Program Module

The example program of this chapter will move three messages which are contained in a data table to another memory location. The three messages are labeled MESSAGE\_1, MESSAGE\_2, and MESSAGE\_3. The ends of the messages are also labeled so that the program will know how many words of data to transfer. The destination memory location is labeled VIDEO\_RAM.

## Getting Started Description of the Example Program

```

XDEF  START,VIDEO_RAM ;External definitions.
XREF  TRANSFER,DELAY  ;External references.

SECT  TABLE          ;Section name.

MESSAGE_1  DC.B  'The example program moves '
            DC.B  'this and two additional '
            DC.B  'messages to a RAM location. '

MESSAGE_1_END
MSG_1_LENGTH EQU  MESSAGE_1_END-MESSAGE_1-2

MESSAGE_2  DC.B  'The first message is '
            DC.B  'displayed for a medium '
            DC.B  'length of time. '

MESSAGE_2_END
MSG_2_LENGTH EQU  MESSAGE_2_END-MESSAGE_2-2
PAGE

MESSAGE_3  DC.B  'The second message is '
            DC.B  'displayed for a shorter '
            DC.B  'length of time. '

MESSAGE_3_END
MSG_3_LENGTH EQU  MESSAGE_3_END-MESSAGE_3-2

START      SECT  M_CODE          ;Section name.
            MOVE  #STACK,A7      ;Initialize user stack.

SET_UP     MACRO  ADDRESS,LENGTH,COUNT ;Macro definition.
            BSR  CLEAR           ;Clears the message destination.
            MOVE  #ADDRESS,A0    ;Address parameter passed in A0.
            MOVE  #LENGTH/2,D0   ;Length parameter passed in D0.
            BSR  TRANSFER        ;
            MOVE  COUNT,D0       ;Count parameter passed in D0.
            BSR  DELAY           ;
            ENDM                ;Macro terminator.
            PAGE

REPEAT     SET_UP  MESSAGE_1,MSG_1_LENGTH,#10 ;
            SET_UP  MESSAGE_2,MSG_2_LENGTH,#7 ; Macro calls.
            SET_UP  MESSAGE_3,MSG_3_LENGTH,#4 ;
            BRA    REPEAT

CLEAR      MOVE  #VIDEO_RAM,A0
            MOVE  #30H,D0          ;Clear 30H words.
AGAIN      MOVE  #2020H,(A0)+      ;ASCII spaces are moved.
            DBEQ  D0,AGAIN
            RTS


VIDEO_RAM  COMMON  DATA          ;Common section name.
STACK     DS.W  0FFH ;Message destination.
          DS.W  1
          END  START ;Execution to begin at START (load address).

```

**Figure 1. The "mov\_mesg.s" Source File**

## Getting Started

### Description of the Example Program



The example program will (1) move the first message to VIDEO\_RAM, where it will be displayed for about 10 seconds, (2) move the second message to VIDEO\_RAM, where it is displayed for about 7 seconds, and (3) move the third message to VIDEO\_RAM, where it is displayed for about 4 seconds. At this point the program will loop back and display the second and third messages, one after the other, repeatedly. The **mov\_mesg.s** source file is shown in figure 1.

#### External Definitions.

The first thing the **mov\_mesg.s** program module does is define the symbols which can be referenced by other program modules. These definitions are made with the XDEF assembler directive. The label VIDEO\_RAM is defined as an external because the **transfer.s** program module will reference the destination memory locations. The label START is defined as an external for program debugging convenience.

#### External References.

The external reference (XREF) assembler directive allows you to use labels which are defined in other program modules. In the **mov\_mesg.s** program module, the BSR TRANSFER and the BSR DELAY instructions use labels which are defined in the **transfer.s** and **delay.s** program modules, respectively. Therefore, TRANSFER and DELAY must be declared as external references.

#### The TABLE Program Section.

The TABLE program section contains the ASCII (by default) bytes of the three messages which are written to the destination memory location. The DC.B assembler directive is used to define the ASCII data. The lengths of the three messages are assigned to labels with the EQU assembler directive.

#### The M\_CODE Program Section.

The executable code of the **mov\_mesg.s** program module is found in the M\_CODE section. After the user stack pointer is loaded, the SET\_UP macro is defined. The three parameters in the macro definition (ADDRESS, LENGTH, and COUNT) are assigned actual values in the macro calls. Each time the macro is called, assembly code is generated which branches to the CLEAR, TRANSFER, and DELAY subroutines. (Parameters are moved into registers before the TRANSFER and DELAY branches.) After the macro is defined, it is called three times. The CLEAR subroutine, which moves ASCII

spaces to the destination memory locations, appears at the end of the M\_CODE program section.



### The DATA Program Section.

Storage locations are defined in the DATA program section with the DS.W assembler directive. The low part of this storage location is the destination of the three messages and is labeled VIDEO\_RAM. The upper addresses of this storage location is for the user stack and is labeled STACK.

### The "transfer.s" Program Module

The main program branches to the subroutine contained in the **transfer.s** program module. The "transfer" subroutine will move the data from the address passed in A0 to the destination memory location VIDEO\_RAM. Notice that the executable code in this module appears in a program section named T\_CODE. Also, notice the external definition of the label TRANSFER (which allows the main program to branch to this label) and the external reference of the label VIDEO\_RAM which was defined in the main program module. The **transfer.s** source file is shown in figure 2.

```
TRANSFER  XDEF    TRANSFER    ;External definition.
AGAIN     XREF    VIDEO_RAM ;External reference.

          SECT    T_CODE    ;Section name.
          MOVE    #VIDEO_RAM,A1
          MOVE    (A0)+,(A1)+ ;Address of message passed in A0.
          DBEQ    D0,AGAIN  ;Message length passed in D0.
          RTS
```

Figure 2. The "transfer.s" Source File

Getting Started  
Description of the Example Program

### The "delay.s" Program Module

The main program branches to the "delay" subroutine contained in the **delay.s** program module. The "delay" subroutine is used to display the various messages for the number of seconds passed in register D0. This program module's executable code is placed in a program section named D\_CODE. Notice the external definition of the DELAY label so that other program modules can refer to this subroutine. The **delay.s** source file is shown in figure 3.

```

                                XDEF    DELAY          ;External definition.
                                SECT    D_CODE          ;Section name.
DELAY                          MOVE    #553,D1
                                MULU   D1,D0          ;Calculate delay count, result in D0.
                                MULU   D1,D0
                                REPEAT
                                SUBQ.L #1,D0          ;Structured control statement.
                                UNTIL.L D0 <EQ> #0
                                RTS
```

Figure 3. The "delay.s" Source File



## Assembling the Program Module Source Files

Assembling program module source files will create object files. The commands to assemble the source files are shown below:

```
$ as68k -l mov_mesg.s > mov_mesg.lis  
$ as68k -l transfer.s > transfer.lis  
$ as68k -l delay.s > delay.lis
```

The **-L** in the commands above causes an assembler listing on the standard output. The default output format will be HP-MRI IEEE 695 relocatable format (.o or .obj extension). The ">" in the commands above redirects the standard output to a file.

Assembler listings for each of the program modules are shown in figures 4 through 6.

```
1993 HPB3641-19300 A.02.00 27Apr93 Copr. HP 1988 Page 1 Wed Apr 28 15:19:19  
  
Command line: as68k -L mov_mesg.s  
Line Address  
1 ; @(SUBID) MAIN: /lsd/nls/proc/680xx/asmlnklib  
0.09 19Apr93 15:03:41 ; MKT:@(#) B3641-19300 A.02.00 68K FAMILY CROSS  
2 ASSEMBLER/LINKER 19Apr93 $  
3 XDEF START,VIDEO_RAM  
;External definitions.  
4 XREF TRANSFER,DELAY  
;External references.  
5  
6 SECT TABLE  
;Section name.  
7  
8 00000000 5468 6520 6578 MESSAGE_1 DC.B 'The example program  
moves '  
616D 706C 6520  
7072 6F67 7261  
6D20 6D6F 7665  
7320
```

Figure 4. The "mov\_mesg.lis" Listing

## Getting Started Assembling the Program Module Source Files

```

9          0000001A 7468 6973 2061          DC.B    'this and two
additional '
          6E64 2074 776F
          2061 6464 6974
          696F 6E61 6C20
10         00000032 6D65 7373 6167          DC.B    'messages to a RAM
location. '
          6573 2074 6F20
          6120 5241 4D20
          6C6F 6361 7469
          6F6E 2E20
11
12         0000004C          MESSAGE_1_END
MESSAGE_1_END-MESSAGE_1-2          MESG_1_LENGTH EQU
13
14         0000004E 5468 6520 6669          DC.B    'The first message is '
          7273 7420 6D65
          7373 6167 6520
          6973 20
15         00000063 6469 7370 6C61          DC.B    'displayed for a medium
'
          7965 6420 666F
          7220 6120 6D65
          6469 756D 20
16         0000007A 6C65 6E67 7468          DC.B    'length of time. '
          206F 6620 7469
          6D65 2E20
17
18         0000003A          MESSAGE_2_END
MESSAGE_2_END-MESSAGE_2-2          MESG_2_LENGTH EQU
19
HPB3641-19300 A.02.00 27Apr93  Copr. HP 1988 Page 2 Wed Apr 28 15:19:19 1993

Line      Address
20
21         0000008A 5468 6520 7365          DC.B    'The second message is '
          636F 6E64 206D
          6573 7361 6765
          2069 7320
22         000000A0 6469 7370 6C61          DC.B    'displayed for a
shorter '
          7965 6420 666F
          7220 6120 7368
          6F72 7465 7220
23         000000B8 6C65 6E67 7468          DC.B    'length of time. '
          206F 6620 7469
          6D65 2E20
24
25         0000003C          MESSAGE_3_END
MESSAGE_3_END-MESSAGE_3-2          MESG_3_LENGTH EQU
26
27
          SECT    M_CODE
;Section name.
28         00000000 3E7C 01FE          R START          MOVE    #STACK,A7

```

Figure 4. The "mov\_mesg.lis" Listing (Cont'd)

## Getting Started Assembling the Program Module Source Files

```

;Initialize user stack.
29
30                               SET_UP      MACRO   ADDRESS,LENGTH,COUNT
;Macro definition.
31                               BSR        CLEAR          ;Clears
the message destination.
32                               MOVE       #ADDRESS,A0
;Address parameter passed in A0.
33                               MOVE       #LENGTH/2,D0   ;Length
parameter passed in D0.
34                               BSR        TRANSFER
35                               MOVE       COUNT,D0       ;Count
parameter passed in D0.
36                               BSR        DELAY
37                               ENDM
;Macro terminator.

```

HPB3641-19300 A.02.00 27Apr93 Copr. HP 1988 Page 3 Wed Apr 28 15:19:19 1993

```

Line      Address
39
40                               SET_UP
MESSAGE_1,MESG_1_LENGTH,#10    ;
40.1      00000004 6100 0048    BSR        CLEAR          ;Clears
the message destination.
40.2      00000008 307C 0000    R          MOVE       #MESSAGE_1,A0
;Address parameter passed in A0.
40.3      0000000C 303C 0026    MOVE       #MESG_1_LENGTH/2,D0
;Length parameter passed in D0.
40.4      00000010 6100 FFE6    E          BSR        TRANSFER
40.5      00000014 303C 000A    MOVE       #10,D0        ;Count
parameter passed in D0.
40.6      00000018 6100 FFE6    E          BSR        DELAY
41                               SET_UP
MESSAGE_2,MESG_2_LENGTH,#7     ; Macro calls.
41.1      0000001C 6100 0030    BSR        CLEAR          ;Clears
the message destination.
41.2      00000020 307C 004E    R          MOVE       #MESSAGE_2,A0
;Address parameter passed in A0.
41.3      00000024 303C 001D    MOVE       #MESG_2_LENGTH/2,D0
;Length parameter passed in D0.
41.4      00000028 6100 FFD6    E          BSR        TRANSFER
41.5      0000002C 303C 0007    MOVE       #7,D0         ;Count
parameter passed in D0.
41.6      00000030 6100 FFCE    E          BSR        DELAY
42                               SET_UP
MESSAGE_3,MESG_3_LENGTH,#4     ;
42.1      00000034 6100 0018    BSR        CLEAR          ;Clears
the message destination.
42.2      00000038 307C 008A    R          MOVE       #MESSAGE_3,A0

```

**Figure 4. The "mov\_mesg.lis" Listing (Cont'd)**

## Getting Started

### Assembling the Program Module Source Files

```

;Address parameter passed in A0.
42.3      0000003C 303C 001E      MOVE      #MESG_3_LENGTH/2,D0
;Length parameter passed in D0.
42.4      00000040 6100 FFB6      E          BSR      TRANSFER
42.5      00000044 303C 0004      MOVE      #4,D0          ;Count
parameter passed in D0.
42.6      00000048 6100 FFB6      E          BSR      DELAY
43        0000004C 60CE          BRA      REPEAT
44
45        0000004E 307C 0000      R CLEAR      MOVE      #VIDEO_RAM,A0
46        00000052 303C 0030      MOVE      #30H,D0
;Clear 30H words.
47        00000056 30FC 2020      AGAIN      MOVE      #2020H,(A0)+
;ASCII spaces are moved.
48        0000005A 57C8 FFFA      DBEQ      D0,AGAIN
49        0000005E 4E75          RTS
50
51          COMMON DATA          ;Common
section name.
52        00000000          VIDEO_RAM DS.W 0FFH      ;Message
destination.
53        000001FE          STACK   DS.W 1
54          END      START      ;Execution to
begin at START (load address).

```

HPB3641-19300 A.02.00 27Apr93 Copr. HP 1988 Page 4 Wed Apr 28 15:19:19 1993

#### Symbol Table

Label	Value
AGAIN	M_CODE:00000056
CLEAR	M_CODE:0000004E
DELAY	External
MESG_1_LENGTH	0000004C
MESG_2_LENGTH	0000003A
MESG_3_LENGTH	0000003C
MESSAGE_1	TABLE :00000000
MESSAGE_1_END	TABLE :0000004E
MESSAGE_2	TABLE :0000004E
MESSAGE_2_END	TABLE :0000008A
MESSAGE_3	TABLE :0000008A
MESSAGE_3_END	TABLE :000000C8
REPEAT	M_CODE:0000001C
SET_UP	Macro
STACK	DATA :000001FE
START	M_CODE:00000000
TRANSFER	External
VIDEO_RAM	DATA :00000000

Figure 4. The "mov\_mesg.lis" Listing (Cont'd)

## Getting Started Assembling the Program Module Source Files

```

1993                HPB3641-19300 A.02.00 27Apr93  Copr. HP 1988 Page 1 Wed Apr 28 15:19:30

Command line: as68k -L transfer.s
Line Address
1                  ; @(SUBID) MAIN: /lsd/nls/proc/680xx/asmlnklib 0.09
19Apr93 15:03:41
2                  ; MKT:@(#) B3641-19300 A.02.00 68K FAMILY CROSS
ASSEMBLER/LINKER  19Apr93
                    $
3                  XDEF    TRANSFER    ;External
definition.
4                  XREF    VIDEO_RAM   ;External
reference.
5
6                  SECT    T_CODE      ;Section name.
7  00000000 327C 0000      E TRANSFER  MOVE    #VIDEO_RAM,A1
8  00000004 32D8          AGAIN      MOVE    (A0)+,(A1)+ ;Address of
message passed in A0.
9  00000006 57C8 FFFC          DBEQ    D0,AGAIN   ;Message length
passed in D0.
10 0000000A 4E75          RTS
11                  END
                HPB3641-19300 A.02.00 27Apr93  Copr. HP 1988 Page 2 Wed Apr 28 15:19:30
1993

```

```

                Symbol Table

Label          Value
AGAIN          T_CODE:00000004
TRANSFER      T_CODE:00000000
VIDEO_RAM     External

```

**Figure 5. The "transfer.lis" Assembly Listing**

## Getting Started Assembling the Program Module Source Files

```

1993                HPB3641-19300 A.02.00 27Apr93  Copr. HP 1988 Page 1 Wed Apr 28 15:18:34

Command line: as68k -L delay.s
Line          Address
1              ; @(SUBID) MAIN: /lsd/nls/proc/680xx/asmlnklib
0.09 19Apr93 15:03:41
2              ; MKT:@(#) B3641-19300 A.02.00 68K FAMILY CROSS
ASSEMBLER/LINKER      19Apr93
                    $
3              XDEF      DELAY
;External definition.
4
5              SECT      D_CODE
;Section name.
6              00000000 323C 0229      DELAY      MOVE      #553,D1
7              00000004 COC1          MULU      D1,D0
;Calculate delay count, result in D0.
8              00000006 COC1          MULU      D1,D0
9              REPEAT
;Structured control statement.
9.1            ??0001 ;> REPEAT <
10             00000008 5380          SUBQ.L   #1,D0
11             UNTIL.L D0 Q #0
11.1           0000000A 0C80 0000 0000  ??0002 CMP.L   #0,D0 ;> UNTIL <
11.2           00000010 66F6          BNE     ??0001 ;> UNTIL <
11.3           ??0003 ;> UNTIL <
12             00000012 4E75          RTS
13             END
                HPB3641-19300 A.02.00 27Apr93  Copr. HP 1988 Page 2 Wed Apr 28 15:18:34
1993

```

### Symbol Table

Label	Value
DELAY	D_CODE:00000000

**Figure 6. The "delay.lis" Assembly Listing**

## Creating an Example Library File

One of the objectives of this chapter was to show how object modules can be linked from libraries. Before we can link from a library file, one must first be created. To create an example library file consisting of the "transfer.o" and "delay.o" relocatable object modules, enter the following command:

```
$ ar68k -a transfer.o,delay.o -L exlib >  
exlib.lis
```

Use ".obj" instead of ".o" if you are using MS-DOS.

The **-a** option in the command above specifies that the files which follow are to be added to the library. The **-L** option in the command above specifies that a library listing file be sent to the standard output (which is redirected to the "exlib.lis" file). The library listing file is shown in figure 7.

## Getting Started Creating an Example Library File

HPB3641-19300 Wed Apr 28 15:19:56 1993

Version A.02.00

Library being built exlib.a

Module	Size	Processor
transfer ...	352	68000

\*\*\*\*\* PUBLIC DEFINITIONS \*\*\*\*\*  
TRANSFER

\*\*\*\*\* EXTERNAL REFERENCES \*\*\*\*\*  
VIDEO\_RAM

Public Count = 1  
External Count = 1

Module	Size	Processor
delay ...	307	68000

\*\*\*\*\* PUBLIC DEFINITIONS \*\*\*\*\*  
DELAY

Public Count = 1  
External Count = 0  
Module Count = 2

**Figure 7. The "exlib.lis" Library Listing**



---

## Linking the Program Module Relocatable Object Files

Linking is the process in which program modules are joined together to form a single absolute file which can then be executed or debugged. Because you can link several object modules to form an executable file, the linking loader is sometimes called the "linker". Also, because you can specify the load addresses of various program sections, the linking loader will sometimes be referred to as the "loader". Either name is correct; the **ld68k** tool does both.

There are two ways that you can specify object files to be linked: (1) you can enter the names of the files on the command line, or (2) you can specify the names of the object files in a linker command file. The linker command file shown in figure 8 will be used to link the three object modules in the example program.

```
NAME demo                      ; Specifies output module name.
LIST C,D,O,P,S,T,X

; List the cross reference (C), place PUBLIC symbols in the output
; object module (D), produce an object module (O - Not necessary,
; this is the default), place input module (local) symbols into the
; Loader symbol table (P - Not necessary, this is the default),
; write local symbol table to the output module (S), list the local
; symbol table (T), and list the PUBLIC symbol table (X).

ORDER M_CODE,T_CODE,D_CODE     ; The T_CODE and D_CODE program sections
                               ; should follow the M_CODE program section.

SECT TABLE=1000H              ; Put the table of messages at 1000H.
SECT M_CODE=1400H              ; Put the M_CODE section at 1400H.
COMMON DATA=1800H            ; Put VIDEO_RAM memory at 1800H.

; Load from these object modules and libraries:
LOAD transfer.o,mov_mesg.o,exlib.a

END                             ; End of linker command file.
```

**Figure 8. The "demo.k" Linker Command File**

### **Linking the Object Modules**

The command to link the example program object modules is shown below. The **-c** option specifies that a linker command file will be supplying information to the linking loader.

```
$ ld68k -L -c demo.k > demo.lis
```

The **-L** option in the command above specifies that an output load map listing file be sent to the standard output (which is redirected to the "demo.lis" file). The output format will be the default HP-MRI IEEE 695 absolute format (.x or .abs extension). The load map listing file is shown in figure 9.

## Getting Started Linking the Program Module Relocatable Object Files

```
HPB3641-19300 A.02.00 27Apr93  Copr. HP 1988  Wed Apr 28 15:20:41 1993
Page 1

Command line: ld68k -L -c demo.k

NAME demo ; Specifies output module name.
LIST C,D,O,P,S,T,X

; List the cross reference (C), place PUBLIC symbols in the output
; object module (D), produce an object module (O - Not necessary,
; this is the default), place input module (local) symbols into the
; Loader symbol table (P - Not necessary, this is the default),
; write local symbol table to the output module (S), list the local
; symbol table (T), and list the PUBLIC symbol table (X).

ORDER M_CODE,T_CODE,D_CODE ; The T_CODE and D_CODE program sections
; should follow the M_CODE program section.

SECT TABLE=1000H ; Put the table of messages at 1000H.
SECT M_CODE=1400H ; Put the M_CODE section at 1400H.
COMMON DATA=1800H ; Put VIDEO_RAM memory at 1800H.

; Load from these object modules and libraries:
LOAD transfer.o,mov_mesg.o,exlib.a

END ; End of linker command file.
HPB3641-19300 A.02.00 27Apr93  Copr. HP 1988  Wed Apr 28 15:20:42 1993
Page 2
```

```
OUTPUT MODULE NAME: demo
OUTPUT MODULE FORMAT: IEEE
```

### SECTION SUMMARY -----

SECTION	ATTRIBUTE	START	END	LENGTH	ALIGN
TABLE	NORMAL DATA	00001000	000010C7	000000C8	2 (WORD)
M_CODE	NORMAL CODE	00001400	0000145F	00000060	2 (WORD)
T_CODE	NORMAL CODE	00001460	0000146B	0000000C	2 (WORD)
D_CODE	NORMAL CODE	0000146C	0000147F	00000014	2 (WORD)
DATA	COMMON	00001800	000019FF	00000200	2 (WORD)

Figure 9. The "demo.lis" Load Map Listing

## Getting Started

### Linking the Program Module Relocatable Object Files

#### MODULE SUMMARY

-----

MODULE	SECTION:START	SECTION:END	FILE
transfer	T_CODE:00001460	T_CODE:0000146B	/users/merff/asm68k/transfer.o
mov_mesg	TABLE:00001000	TABLE:000010C7	/users/merff/asm68k/mov_mesg.o
	M_CODE:00001400	M_CODE:0000145F	
	DATA:00001800	DATA:000019FF	
delay	D_CODE:0000146C	D_CODE:0000147F	/users/merff/asm68k/exlib.a L

#### LOCAL SYMBOL TABLE

-----

SYMBOL	ATTRIB	SECTION	OFFS/ADDR	MODULE:FUNCTION
AGAIN	ASMVAR	T_CODE	00001464	transfer:
MSG_2_LENGTH	ASMVAR	ABSCONST	0000003A	mov_mesg:
STACK	ASMVAR	DATA	000019FE	mov_mesg:
CLEAR	ASMVAR	M_CODE	0000144E	mov_mesg:
MSG_3_LENGTH	ASMVAR	ABSCONST	0000003C	mov_mesg:
REPEAT	ASMVAR	M_CODE	0000141C	mov_mesg:
MSG_1_LENGTH	ASMVAR	ABSCONST	0000004C	mov_mesg:
MESSAGE_1	ASMVAR	TABLE	00001000	mov_mesg:
MESSAGE_1_END	ASMVAR	TABLE	0000104E	mov_mesg:
MESSAGE_2	ASMVAR	TABLE	0000104E	mov_mesg:
MESSAGE_2_END	ASMVAR	TABLE	0000108A	mov_mesg:
MESSAGE_3	ASMVAR	TABLE	0000108A	mov_mesg:
MESSAGE_3_END	ASMVAR	TABLE	000010C8	mov_mesg:
AGAIN	ASMVAR	M_CODE	00001456	mov_mesg:

#### PUBLIC SYMBOL TABLE

-----

HPB3641-19300 A.02.00 27Apr93 Copr. HP 1988 Wed Apr 28 15:20:42 1993  
 Page 3

SYMBOL	SECTION	ADDRESS	MODULE
DATA	DATA	00001800	\$\$
DELAY	D_CODE	0000146C	delay
START	M_CODE	00001400	mov_mesg
TRANSFER	T_CODE	00001460	transfer
VIDEO_RAM	DATA	00001800	mov_mesg

Figure 9. The "demo.lis" Load Map Listing (Cont'd)

Getting Started  
**Linking the Program Module Relocatable Object Files**

CROSS REFERENCE TABLE  
-----

SYMBOL	SECTION	ADDRESS	MODULE
DATA	DATA	00001800	-\$
DELAY	D_CODE	0000146C	-delay mov_mesg
START	M_CODE	00001400	-mov_mesg
TRANSFER	T_CODE	00001460	-transfer mov_mesg
VIDEO_RAM	DATA	00001800	-mov_mesg transfer

START ADDRESS: 00001400

Link Completed

**Figure 9. The "demo.lis" Load Map Listing (Cont'd)**

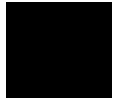
This completes the "Getting Started" example. For a brief description of the **as68k**, **ld68k**, and **ar68k** syntax and options, refer to the "Command Syntax" chapter which follows.

Getting Started  
**Linking the Program Module Relocatable Object Files**



---

# 2



---

## Command Syntax

This chapter contains the on-line manual pages, which briefly describe the syntax for using the assembler, linker, and librarian.

## Command Syntax

Options may be entered on the command line to control generation of the output listing and object module, and to turn internal assembler flags on and off.

The command syntax information in this chapter may also be found in the on-line manual pages:

- If you are using a PC, look for the **.txt** files in the assembler directory.
- If you are using a UNIX system, use the **man** command. For example, to view the **as68k** on-line manual page, just type in the following command from your operating system prompt:

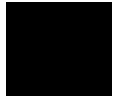
```
$ man as68k 
```

Information on the **as68k** assembler syntax and options will be scrolled onto your display.



## File Extensions

UNIX Extension	DOS Extension	Meaning	Where generated
<b>.a</b>	<b>.lib</b>	Archive (library) file	ar68k
<b>.A</b>	<b>.a</b>	HP 64000 format assembler symbol file	as68k
<b>.k</b>	<b>.k</b>	Linker command file (default extension used by cc68k)	editor
<b>.L</b>	<b>.l</b>	HP 64000 format linker symbol file	ld68k -h
<b>.l</b>	<b>.txt</b>	On-line manual page	Provided
<b>.o</b>	<b>.obj</b>	HP-MRI IEEE-695 format relocatable object file	as68k
<b>.O</b>	<b>.lst</b>	Listing file	cc68k -L
<b>.s</b>	<b>.s</b>	Assembly language source file	cc68k or editor
<b>.x</b>	<b>.abs</b>	HP-MRI IEEE-695 format or Motorola S-Record absolute object file (executable)	ld68k
<b>.X</b>	<b>.x</b>	HP 64000 format absolute file (executable)	ld68k -h



## **as68k(1)**

**NAME** as68k - cross assembler for Motorola family microprocessors

**SYNOPSIS** **/usr/hp64000/bin/as68k** [options] [file]  
**/usr/hp64000/bin/as68030** [options] [file]  
**/usr/hp64000/bin/as68040** [options] [file]

Under DOS on a PC:  
**\hpa68k\as68k** [options] [file]

**DESCRIPTION** The *as68k* command assembles the named *file*, or the standard input if no file name is specified. If no file name is specified, the names of output files must be specified explicitly using options.

If *as68030* or *as68040* is used to invoke the assembler, the default chip is set to 68030 or 68040.

The *as68k* program first attempts to open *file* for reading. If that fails, the assembler appends *.s* and attempts to open *file.s*. Under DOS on a PC the string *.src* is appended rather than *.s*.

If no input file is specified and standard input is a tty, *as68k* displays a usage diagnostic and terminates.

The output is a relocatable file containing Motorola microprocessor instructions and symbolic data. The format of the output file is HP-MRI IEEE-695 (HP's implementation of the IEEE 695 MUFOM format). If no output file is specified (using **-o**), the pathname and the ending suffix are stripped from the input file name and **.o** is appended to it. Under DOS, **.obj** will be appended rather than **.o**. This becomes the name of the output file.

The following options are recognized by *as68k*:

**-b**

Big. This option allows very large source files to be assembled. Normally, for the sake of speed, intermediate data (whose size is proportional to the size of the source file) is kept in virtual memory. The **-b** option causes intermediate data to be stored in temporary files on the host file system. Use this option if **ERROR (604): Out of virtual memory** occurs.

**-D** *name*

**-D** *name=def*

Define *name* as if by a `\# define \C` language directive. If no `\=def` is given, *name* is defined as 1.

**-f** *flaglist*

The flags in *flaglist* are used to select and change the internal assembler control switches. The flags recognized and their meanings are defined below. A more complete explanation may be found in the *HP 68000 Family Assembler/Linker/Librarian User's Guide*. Each flag may be set (or unset) in either of two ways. A flag may be set on the command line using **-f** option as described here. A flag may also be set by using the **OPT** pseudo-operator in the assembler source program. Groups of flags following the **-f** option must be separated by commas or separated by white space and quoted. For example, the following sets the flags *brs*, *d*, *p=68000*, and *x*.

```
-f brs,d -f "p=68000 x"
```

A flag may be unset (turned off) by preceding the flag value with **-** or **no**. For example, the following turns off the *abspcadd* and *o* flags.

```
-f noabspcadd,-o
```

**-H** *asmb\_sym\_file*

This option overrides the default file name for the HP 64000 format assembler symbol file. (See the **-h** option below.) If *asmb\_sym\_file* has a suffix, then the name is used as is. Otherwise, **.A** is appended to form *asmb\_sym\_file.A*.

**-h**

This option indicates that the assembler should produce an HP 64000 format assembler symbol file for debugging purposes. The default name for the assembler symbol file is *file.A*. *File* is the source file name with any preceding directories and trailing suffix stripped off. The default assembler symbol file name may be overridden with the **-H** option. When writing the *asmb\_sym* file, all identifiers in the source program are converted to legal HP64000 identifiers. That is, Motorola assembly language identifiers may contain the characters **.** (period), **?** (question mark), and **\$** (dollar sign) and have a maximum of 31 significant characters. To produce legal HP64000 identifiers; all periods, question marks, and dollar signs are converted to **\_** (underbar) and identifiers are truncated to 15 characters maximum.

## Command Syntax as68k(1)

### **-I** *directory*

The assembler searches *directory* for INCLUDE files. The assembler first looks in the present working directory and then in up to four additional directories specified by the **-I** options.

### **-l**

specifies that an assembler listing file be written to standard output. This listing contains offsets, instruction codes, symbol table information, symbol table cross reference, and other useful information.

### **-o** *objfile*

specifies the name of the output file. This overrides the default file name for the HP-MRI IEEE-695 relocatable file produced. If *obj\_file* has a suffix, then the name is used as is. Otherwise, **.o** is appended to form *obj\_file.o*. On a PC running DOS, the default extension is **.obj**.

## FLAGS

The following flags may be specified using the **-f flaglist** option. In some cases there are several spellings for the same flag.

**abspcadd** Causes the operand *6(PC)* to be interpreted as a reference to the absolute address 6. Unsetting the flag (**noabspcadd**) causes the above operand to be interpreted as a displacement of 6 from the PC. (default: **abspcadd**)

**b, brb, brs** Forces 8-bit displacements in branch instructions (Bcc, BRA, BSR) to forward locations. Explicit qualifiers (e.g. BRA.L) override this flag. (default: **brw**)

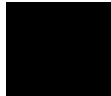
**brw** Forces 16-bit displacements in branch instructions (Bcc, BRA, BSR) to forward locations. Explicit qualifiers (e.g. BRA.L) override this flag. (default: **brw**)

**brl** Forces 32-bit displacements in branch instructions (Bcc, BRA, BSR) to forward locations when supported by the processor and when **noold** is in effect. When **old** is in effect, then **brl** forces 16-bit displacements. For all other processors, this forces 16-bit displacements. Explicit qualifiers (e.g. BRA.L) override this flag. (default: **brw**)

**c, cex** List all lines of object code (after the first) generated by the DC pseudo-op. (default: **c**)

**case** User defined symbols are case sensitive. **Nocase** means that upper and lower case letters in identifiers are equivalent. (default: **case**)

- d** Place the symbol table in the object module. (default: **d**)
- e** List lines with errors and warnings to standard error. (default: **e**)
- f, frs** Causes assembler to allocate 16 bits for operand extensions for operands of the form *expression* where *expression* contains a forward reference. During pass 2, the assembler may decide to access the operand using absolute-short or PC-plus-displacement modes. (default: **frl**)
- frl** Causes assembler to allocate 32 bits for operand extensions for operands of the form *expression* where *expression* contains a forward reference. During pass 2, the assembler may decide to access the operand using absolute-short, absolute-long, or PC-plus-displacement modes. (default: **frl**)
- g** List assembler-generated symbols in the symbol or cross reference listing. If **d** is also set, these symbols are placed in the object module as well. (default: **nog**)
- hlsym** Affects the symbolic information in the IEEE relocatable file for compiler-generated modules. **Hlsym** causes assembly-level local symbols to be put into the output file. **Nohlsym** omits assembly-level local symbols from compiler-generated modules resulting in smaller output files. Compiler-generated symbols are not affected by this flag. (default: **nohlsym**)
- i, cl** List instructions not assembled due to conditional assembly statements. (default: **i**)
- m, mex** List macro and structured control directive expansions in program listing. (default: **mex**)
- mc** List macro calls in program listing. (default: **mc**)
- md** List macro definitions in program listing. (default: **md**)
- o** Produce an output relocatable module. (default: **o**)
- old** Specifies that the interpretation of the **brl** flag and explicit **.L** qualifiers on Bcc instructions will be 16-bit displacements (as appropriate for the 68010 and earlier processors), even though the processor mode has been set to indicate a processor with an address bus width greater the 16 bits. This flag is useful when migrating 68000 programs. (default: **noold**)
- opnop** Remove NOP instructions generated by the assembler. When the assembler encounters a forward reference during pass 1, it will allocate space for an instruction based on worst case assumptions. During pass 2, it will



## Command Syntax as68k(1)

sometimes generate a shorter form of the instruction and fill the remaining space with NOPs. This flag removes those NOPs but at the cost of increased assembly time because it makes additional passes over the file. (default: **noopnop**)

**p= proc** Identifies the target processor (default: 68000). Valid values for *proc* are: **68000 68EC000 68HC000 68HC001 68302 68008 68010 68330 68331 68332 68333 68340 CPU32 68020 680EC20 68030 680EC30 68040 68EC040**

**p, pco** Assembler uses PC-plus-displacement mode to access operands (of the form *expression*) within an absolute section. **Nopco** causes such references to use absolute mode. (default: **nopco**)

**pcr** Assembler uses PC-plus-displacement mode to access operands (of the form *expression*) within a relocatable section. **Nopcr** causes such references to use absolute mode. (default: **pcr**)

**quick Quick** allows the assembler to optimize certain mnemonics when possible. The mnemonics are: MOVE to MOVEQ, ADD to ADDQ, and SUB to SUBQ. **Noquick** prevents these optimizations. (default: **quick**)

**r**

**pcs** Assembler uses PC-plus-displacement mode to access operands (of the form *expression*) when the instruction is in a relocatable section and the operand is in a different relocatable section. **Nopcs** causes such references to use absolute mode. (default: **nopcs**)

**rel32** This flag applies to the following 68020 address modes.

(bd, An, Xn) ([bd, An, Xn], od) ([bd, An], Xn, od)  
(bd, PC, Xn) ([bd, PC, Xn], od) ([bd, PC], Xn, od)

**Rel32** causes the assembler to use 32-bit base and outer displacements for forward, external, or relocatable operands. **Norel32** causes 16-bit base and outer displacements. This flag applies to operands that do not have explicit word or longword size qualifiers. (default= **norel32**)

**s** List the source text in program listing. (default: **s**)

**t** List the symbol table in program listing. (default: **t**)

**w** Generate messages for warnings. **now** means suppress warnings. (default: **w**)

**x, cre** List the cross reference table in the program listing. The cross reference table replaces the symbol table in the listing. (default: **nox**)

**FILES** file.s Assembly language source file. (Unix)  
file.src Assembly language source file. (DOS)  
file.o HP-MRI IEEE-695 relocatable object file. (Unix)  
file.obj HP-MRI IEEE-695 relocatable object file. (DOS)  
file.A HP 64000 format assembler symbol file

**SEE ALSO** *HP 68000 Family Assembler/Linker/Librarian User's Guide*, ld68k(1), ar68k(1).

**DIAGNOSTICS** The *as68k* command returns zero if no errors are detected in the assembly source. Otherwise, it returns non-zero.

Diagnostic messages including optional lines containing assembly errors are displayed on standard error.

**BUGS** The following is not a defect but rather a sometimes misunderstood aspect of *as68k*.

Beware of labels on a line by themselves. They may not be aligned as you expect. For example,

```

                SECT    A
STRING DC.B    'odd'
START
                LEA    STACKTOP, SP

```

The label *START* will have an odd value. If the PC is loaded with an odd value, a run time error will occur.

There are two ways to avoid this problem. You may put the label on the same line as the instruction or directive. The label will have the same alignment as the instruction. For example,

```

                SECT    A
STRING DC.B    'odd'
START LEA    STACKTOP, SP

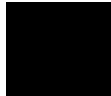
```

You may also use an *align* directive after the byte constants. For example,

```

                SECT    A
STRING DC.B    'odd'
                ALIGN  2
START
                LEA    STACKTOP, SP

```



## **ld68k(1)**

**NAME** ld68k - cross linker/loader for Motorola family microprocessors

**SYNOPSIS** **/usr/hp64000/bin/ld68k** [options] [files]  
**/usr/hp64000/bin/ld68030** [options] [files]  
**/usr/hp64000/bin/ld68040** [options] [files]

Under DOS on a PC:

**\hpas68k\ld68k** [options] [files]

**DESCRIPTION** The *ld68k* command takes one or more relocatable object files as input and combines them to produce a single output file. In doing so it resolves references to external symbols, assigns final addresses to procedures and variables, revises code and data to reflect new addresses and updates symbolic debug information (when it is present in a file).

If *ld68030* or *ld68040* is used to invoke the linker the default chip is set to 68030 or 68040.

By default, the output file format is HP-MRI IEEE-695 (HP's implementation of the IEEE standard 695 MUFOM format). This file contains Motorola 680xx instructions and symbolic data. Options may be used to create output files in HP 64000 format or Motorola S-Record format. Refer to the OUTPUT FILE FORMATS section which follows for more information.

Usually, the output file contains instructions and data in absolute form. That is, address information has been supplied to locate the program in target memory.

The **-i** option may be used to specify a relocatable output file in a process called *incremental linking*. In an incremental link, the input relocatable files are simply combined into an output relocatable file. While address information may be specified in an incremental link, the instructions and data remain in relocatable form. The addresses specified during an incremental link may be changed in subsequent links.

The operation of *ld68k* is controlled by LINKER COMMANDS (described below). Linker commands specify the input relocatable and archive files, the location and order of relocatable sections, and the contents of the output files.



The *ld68k* program reads commands from the command line or from a *command\_file* using either the **-c** option. The *ld68k* program no longer reads commands from standard input using pipes or interactive commands.

The *ld68k* program accepts relocatable and archive input files in HP-MRI IEEE-695 format. These files may be produced by the cross compiler, the cross assembler (*as68k*), the cross linker itself (*ld68k*), or the archive file librarian (*ar68k*).

Input files may be specified in "LOAD" commands or on the command line. The order of specification of the input files is significant to the operation of the linker.

If input files are specified on the command line, these files are loaded in addition to files specified in "LOAD" commands in the *command\_file*. Input files specified on the command line will precede any input files mentioned in LOAD commands.

If the input file names have a suffix, then the name is used as is. Otherwise, *ld68k* appends **.o** to the name on the command line. The suffix **.obj** is appended when *ld68k* is run on a PC under DOS.

The name of the output file may be specified with the **-o** option; if that is omitted, the name of the output file is derived from the name of the *command\_file*. It is an error if neither the output file name nor the *command\_file* name is specified.

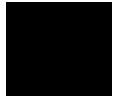
The default names of the output files are determined in the following way. Any pathname and any ending suffix (i.e. including the last '.') is stripped from the *command\_file* name to form the basic output file name. Then, depending on the type of the output file, a suffix is appended to the basic name to form the output file name.

If the output is HP-MRI IEEE-695 absolute format, then the suffix is **.x**. The suffix will be **.abs** on the PC host.

If the output is HP 64000 format, then the suffixes are **.X** for the absolute file and **.L** for the linker symbol file.

If the output is Motorola S-Record format, then the suffix is **.x**.

If an incremental link is done, then the output is in HP-MRI IEEE-695 relocatable format and the suffix is **.o**. On the PC host running DOS, the suffix will be **.obj**.



## Command Syntax ld68k(1)

The following options are recognized by *ld68k*:

### **-b**

This option is included for backward compatibility and does not affect *ld68k* operation.

### **-c** *command\_file*

This option specifies the name of the *command\_file* to be used to supply information to *ld68k*. The file name part of the command file path, with suffix stripped, is used to form the default names of output files.

### **-f** *flaglist*

This linker command will **NOT** be present in future versions. All flag functionality will be accessible via other command line options and/or the linker command file. The flags in *flaglist* are used to select and change the internal linker control switches. The flags recognized and their meanings are defined below. A more complete explanation may be found in the *HP 68000 Family Assembler/Linker/Librarian User's Guide*. Each flag may be set (or unset) in either of two ways. A flag may be set on the command line using the **-f** option described here. A flag may also be set using the **LIST** linker command and unset using the **NLIST** linker command. Groups of flags following the **-f** option must be separated by commas or separated by white space and quoted. For example, the following option sets the flags *c*, *d*, *s*, and *x*.

```
-f c,d -f "s x"
```

A flag may be unset (turned off) by preceding the flag with **-** or **no**. For example, the following option turns off the *o* and *p* flags.

```
-f noo,-p
```

Errors in the *flaglist* are not detected immediately when the command line is processed. Rather, the loader acts as if a "LIST *flaglist*" command preceded the first command in the loader command file.

### **-H** *link\_sym\_file*

This option overrides the default file name for the HP 64000 format linker symbol file. (See the **-h** option below.) If *link\_sym\_file* has a suffix, then the name is used as is. Otherwise, **.L** is appended to form *link\_sym\_file.L*.

**-h**

The option indicates that the linker should produce HP 64000 format output files. There are two output files, the absolute file and the linker symbol file. The default name for the absolute file is *command\_file.X* while the default name for the linker symbol file is *command\_file.L*. When writing the link\_sym file, all identifiers (i.e. global symbol definitions) are converted to legal HP64000 identifiers. That is, Motorola assembly language identifiers may contain the characters . (period), ? (question mark), and \$ (dollar sign) and have a maximum of 31 significant characters. To produce legal HP64000 identifiers in the link\_sym file, all periods, question marks, and dollar signs are converted to \_ (underbar) and identifiers are truncated to 15 characters maximum.

**-i**

Specifies that an incremental link be performed. The relocatable input files are combined to produce a relocatable output file. The name of the relocatable output file defaults to *command\_file.o*. On a PC machine running DOS, the file name defaults to *command\_file.obj*. The following linker commands are illegal during an incremental link: ABSOLUTE, BASE, CPAGE, INDEX, INITDATA, NOPAGE, ORDER, PAGE, RESADD, RESMEM, and SORDER.

**-L**

Specifies that output load map listing be written to standard output.

**-m**

Same as **-L** above.

**-o** *objfile*

Specifies the name of the output file. This overrides the default file name for HP-MRI IEEE-695 absolute file, the HP-MRI IEEE-695 relocatable file, the HP 64000 format absolute file, or the Motorola S-Record file. If *obj\_file* has a suffix, then the name is used as is. Otherwise the appropriate suffix will be appended.

**-u** *symbol*

Creates an external reference to *symbol*. This reference may force the linker to load a library module. The EXTERN command performs the same function as the **-u** option.

Command Syntax  
**Id68k(1)**

LINKER  
COMMANDS

The linker/locator recognizes the following commands. Square brackets [] enclose optional parameters. Ellipsis ... indicate the preceding item may be repeated.

*; comment text ...*

Designates a comment.

*# command continuation character*

Allows a command to be continued on the following line.

*' escape character*

Causes the character following the escape char to be treated as a normal character.

**ABSOLUTE** *sectname [,sectname] ... c*

auses only the code from the specified relocatable *sectname(s)* to be written to the absolute output file. Without the **ABSOLUTE** command, code from all absolute and relocatable sections is written. See **LOAD\_SYMBOLS** command.

**ALIAS** *sectname1,sectname2*

specifies that the code in relocatable section *sectname2* be treated as if it were actually in relocatable section *sectname1*.

**ALIGN** *section= number*

**ALIGNMOD** *section= number*

The **ALIGN** command sets the alignment of the beginning of the section only. *Number* must be a power of 2. The **ALIGNMOD** command increases the alignment boundary of each individual module section to *number*.

**BASE** *address*

specifies the address where the linker begins placing relocatable sections. The **SECT** or **COMMON** commands may override **BASE** for individual sections. *Address* is decimal unless preceded by \$ for hexadecimal, @ for octal, or % for binary.

**CASE** [*class,...*]  
**LOWERCASE** [*class,...*]  
**UPPERCASE** [*class,...*]

control the case sensitivity of various classes of symbols during linking. *Class* may be PUBLICS (to indicate global or external symbols), MODULES (to indicate module names), or SECTIONS (to indicate section names). If no class is specified, all symbol classes are affected. CASE means that upper and lower case characters remain distinct and unchanged. LOWERCASE shifts all letters to lower case and UPPERCASE shifts all letters to upper case.

**CHIP** *processor[,buswidth]*

specifies the target processor. *Processor* may be **68000**, **68EC000**, **68HC000**, **68HC001**, **68008**, **68010**, **68302**, **68330**, **68331**, **68332**, **68333**, **68340**, **68020**, **68EC020**, **68030**, **68EC030**, **68040**, or **68EC040**. The optional *buswidth* is a number specifying the width (in bits) of the address bus of the target system.

**COMMON** *sectname= address*

specifies the load address of a common section. See **BASE** for *address* syntax.

**CPAGE** *sectname*

specifies that the starting address of the common section named *sectname* be rounded up to a \$100 (hexadecimal) boundary.

**DEBUG\_SYMBOLS**  
**NODEBUG\_SYMBOLS**

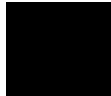
These commands control putting local symbols into output files. These commands may be placed between LOAD commands to selectively copy symbols from certain modules. DEBUG\_SYMBOLS is a synonym for the LIST P command and NODEBUG\_SYMBOLS is a synonym for the NLIST P command.

**END**

Causes the load to be finished and an output module produced.

**ERROR** *condition[,condition] ...*  
**NOERROR** *condition[,condition] ...*  
**WARN** *condition[,condition] ...*

Cause the condition(s) specified to be modified. Condition may be most error or warning numbers. Fatal error conditions may not be modified.



### EXIT

Causes the linker to exit without finishing the load and without producing an output module.

**EXTERN** *name* [*,name*] ...

Creates an external reference to *name*. This reference can cause the loading of a library module.

**FORMAT** *option*

Specifies the format of the object file. *Option* may be **HP** (for HP 64000), **S** (for Motorola S-Record), **IEEE** (for HP-MRI IEEE-695 absolute), **IEEE INCREMENTAL**, or **NOABS** (for no output file). **IEEE INCREMENTAL** is the same as the **-i** option. Default is to produce HP-MRI IEEE-695 absolute. **INCLUDE** *filename* Includes the contents of *filename* in the linker command file.

**INDEX** *?areg,sectname,offset*

Associates an address register with a relocatable section and an offset for the purpose of computing displacements in address-register-plus-displacement mode. The *areg* value may be any of **A2**, **A3**, **A4**, or **A5**.

**INITDATA** *merge\_arg* [*,merge\_arg*] ...

Provides a means of placing one or more initialized data sections in ROM. A section named **??INITDATA** is written to the absolute file. At run time, the sections named by the **??INITDATA** must be moved from the ROM location to their actual link time addresses by an initcopy routine. **??INITDATA** and sections named by **??INITDATA** are ordered and assigned an address using standard linker commands. See **INITDATA** under the **LINKER COMMANDS** section of the user's manual for more information. For a demonstration and sample code see /usr/hp64000/demo/languages/B3641/features/INITDATA. On the PC host, the example code is placed in the examples subdirectory.

**INTFILE**

**NOINTFILE**

INTFILE allows very large programs to be linked. Intermediate data is kept in a temporary file rather than virtual memory. The INTFILE command is equivalent to the **-b** command line option.

**LIST** *flag* [*,flag*] ...

Sets linker flags. The flags may also be set on the command line and are defined below. LIST and NLIST will not be supported in future releases.

**LISTABS** *option* [*,option*] ...

Controls putting different types of symbol information into the output file. LISTABS PUBLICS is the same as LIST D and puts global symbols into S-Record files. LISTABS NOPUBLICS is the same as NLIST D and turns off global symbols. LISTABS INTERNALS is the same as LIST S and puts local symbols in S-record files. LISTABS NOINTERNALS is the same as NLIST S and turns off local symbols to S-Record files. LISTABS NOINTERNALS also turns off all compiler generated symbols and local assembly symbols to IEEE-695 files.

**LISTMAP** *option* [*,option*] ...

Controls the output of certain types of information to the linker listing. The *option* value may be any of CROSSREF, NOCROSSREF, INTERNALS, NOINTERNALS, PUBLICS, or NOPUBLICS. LISTMAP CROSSREF is the same as LIST C and turns on the cross reference listing. LISTMAP INTERNALS is the same as LIST T and turns on the local symbol listing. LISTMAP PUBLICS is the same as LIST X and turns on the global symbol listing.

**LOAD** *filename* [*,filename*] ...

Specifies the name of IEEE relocatable files or archive files from which symbols and code are to be included in the load.

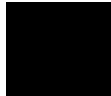
**LOAD\_SYMBOLS** *filename* [*,filename*] ...

Specifies the name of IEEE relocatable files or archive files from which to load symbols and allocate space, code is not loaded. See ABSOLUTE command.

**MERGE** *sectname merge\_arg* [*,merge\_arg*] ...

Renames the sections specified in *merge\_arg* to *sectname*. The MERGE command allows you to select pieces of a section defined in particular modules, change the name of these pieces, and then locate these pieces using the new name. *Merge\_arg* may be any of the following.

*sect2* or {*sect2,module*} or {\*,*module*}



## Command Syntax Id68k(1)

The first form renames all of *sect2* to *sectname*. The second form renames just the portion of *sect2* defined in *module* to *sectname*. The third form renames all the sections defined in *module* to *sectname*.

**NAME** *name*

Specifies the name to be put into the (extended) Motorola S-Record output file.

**NLIST** *flag* [*flag*] ...

Unsets linker flags. Flags are defined below. NLIST and LIST will not be supported in future releases.

**ORDER** *sectname* [,*sectname*] ...

Specifies the order in which ordinary (non-basepage) relocatable sections are placed in memory. The default order is the order in which section names are encountered by the linker, either in linker commands or in input modules.

**PAGE** *sectname*

**NOPAGE** *sectname*

PAGE turns on page relocation (i.e. locating each subsection on a \$100 (hexadecimal) boundary) for *sectname*. NOPAGE restores normal subsection alignment. Default is no page relocation.

**PUBLIC** *name*=*address*

**PUBLIC** *name*=*name2*

Defines a global identifier *name* whose value is either *address* or the value of another symbol *name2*. See **BASE** for *address* syntax.

**RESADD** *low\_addr,high\_addr*

**RESMEM** *low\_addr,size*

Reserve areas of memory that will not be used by the linker for other sections.

**SORDER** *sectname* [,*sectname*] ...

Specifies the order in which short (basepage) relocatable sections are placed in memory.

**SECT** *sectname*=*address*

Specifies the load address of ordinary relocatable section *sectname*. See **BASE** for *address* syntax.



**SECTSIZE** *sect= size*

Allows modification of section size at link time.

**START** *address*

Specifies the starting address for the program. See **BASE** for *address* syntax.

## FLAGS

The following flags may be specified using the **-f flaglist** option.

**a** Produce the output file in Motorola S-Record format. Same as **FORMAT S** command. (default: HP-MRI IEEE-695 format)

**c** Print the identifier cross reference table in the load map. Same as **LISTMAP CROSSREF** command. (default: **noc**)

**d** Put global symbols into the S-Record output file. This flag has no effect on IEEE-695 or HP 64000 files. Same as **LISTABS PUBLICS** command. (default: **nod**)

**h** Produce the output file in HP 64000 format. Same as **FORMAT HP** command. (default: HP-MRI IEEE-695 format)

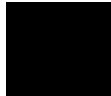
**i** Produce the output file in HP-MRI IEEE-695 format. Same as **FORMAT IEEE** command. (default: HP-MRI IEEE-695 format)

**o** Produce an output file. **LIST NOO** is the same as **FORMAT NOABS** command. (default: **o**)

**p** Place symbols in the input modules into the linker symbol table. This flag affects only Motorola S-Record output files. Its purpose is to exclude symbols from certain input modules from the output module. One does this by surrounding **LOAD** commands with **NLIST P** and **LIST P** commands. Same as **DEBUG\_SYMBOLS** command. (default: **p**)

**s** Put symbols into the output file. The exact behavior depends on the output file format. Same as **LISTABS INTERNALS** command. (default: **s**)

S-Records: **S** writes local symbols and their values in a simple, displayable format at the beginning of the file. **Nos** suppresses these symbols. IEEE-695: **S** writes local assembly symbols and compiler generated symbol and type information to the output file. **Nos** suppresses this information. Global assembly symbols (those mentioned in **XDEF** directives) are always written to the output file regardless of any flag. HP 64000: The **s** flag has no effect on the HP 64000 `link_sym` file.



Command Syntax  
**ld68k(1)**

**t** Print local symbols in the load map. Same as LISTMAP INTERNALS command. (default: **not**)

**x** Print global symbols defined in PUBLIC commands in the load map. Same as LISTMAP PUBLICS command. (default: **nox**)

OUTPUT FILE  
FORMATS

HP 64000 HP 64000 files are consumed by a number of HP 64000 emulators, logic analyzers, and other products. Check the operating manual for your particular HP product to determine what formats it will accept. The HP 64000 absolute, link\_sym, and asmb\_sym file formats are documented in *HP-UX File Format Operating Manual* With this information, you can write your own tools that use the loader's absolute and symbolic output.

HP-MRI IEEE-695 Hewlett Packard's implementation of IEEE 695 MUFOM is consumed by HP 64000 emulators, debuggers, and other products. Check the operating manual for your particular HP product to determine what formats it will accept. Documentation for this format can be obtained by contacting Hewlett Packard.

Motorola S-Records S-Records are used by many non-HP tools. The format expresses absolute code and (optionally) symbol-value pairs using only displayable ASCII characters and newlines. S-Records are described in *HP 64888 File Format Converter Operating Manual*. With this information, you can write your own tools that use the loader's absolute and symbolic output.

FILES

command\_file.x HP-MRI IEEE-695 absolute object file or Motorola S-Record absolute file (Unix)

command\_file.abs HP-MRI IEEE-695 absolute object file or Motorola S-Record absolute file (DOS)

command\_file.X HP 64000 format absolute file

command\_file.L HP 64000 format linker symbol file

command\_file.o HP-MRI IEEE-695 relocatable object file from incremental link (Unix)

command\_file.obj HP-MRI IEEE-695 relocatable object file from incremental link (DOS)

SEE ALSO

*HP 68000 Family Assembler/Linker/Librarian User's Guide*, ar68k(1), as68k(1).

DIAGNOSTICS

The *ld68k* command returns zero if no errors are detected while linking, otherwise returns non-zero.

Diagnostic messages are displayed on standard error.

## BUGS

Programs that linked without error using version 1.00 of *ld68k* may produce **undefined symbol** errors using later versions of *ld68k*. The information below explains the cause of the problem and tells how to correct it.

*As68k* allows identifiers to contain period (.), dollar sign (\$), and question mark (?) and have up to 31 significant characters. Identifiers in HP 64000 *asmb\_sym* and *link\_sym* files may not contain periods, dollar signs, or question marks and can have only 15 significant characters. To create legal HP 64000 identifiers, period, dollar, and question mark are changed to underbar (\_) and identifiers are truncated to 15 characters if necessary.

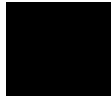
Version 1.00 of *as68k* and *ld68k* differ from later versions with respect to when the conversion was done.

Version 1.00 tools performed Motorola-to-HP symbol conversion only when the **-h** option was used and then immediately when the symbol was seen. Thus, with the **-h** option, a name spelled "a.b\$" would match a name spelled "a\_b\_". This would apply during assembly and/or during linking when global definitions were matched with external references. We thought this was an undesirable and confusing side effect of the **-h** option.

Later versions of the tools never change a symbol's spelling for the purpose of symbol matching. Symbols are converted only when they are written to the HP *asmb\_sym* and *link\_sym* files. Undefined symbols occur because now symbols must always be spelled exactly the same in order to match.

We recommend the following procedure. First, if possible, reassemble all modules that were produced with version 1.00 of *as68k*. Second, after linking, correct undefined symbol errors by going back to the source and changing symbols so that definitions and references are spelled exactly the same.

Version 1.60 linker command file syntax differs somewhat from earlier versions. Most users will need to make changes to pre-1.60 linker command files to use the new INITDATA and comment syntax. See **INITDATA** and ";" under **LINKER COMMANDS**.



## **ar68k(1)**

**NAME** ar68k - archive and library maintainer for Motorola 68k processors.

**SYNOPSIS** **/usr/hp64000/bin/ar68k**  
**/usr/hp64000/bin/ar68k** [**options**] [**action**] ... *archivefile*  
**/usr/hp64000/bin/ar68030** [**options**] [**action**] ... *archivefile*  
**/usr/hp64000/bin/ar68040** [**options**] [**action**] ... *archivefile*

Under DOS on a PC:

**\hpas68k\ar68k** [**options**] [**action**] ... *archivefile*

**DESCRIPTION** The *ar68k* command maintains groups of relocatable files combined into a single archive (or library) file. The archive files may then be used by *ld68k* (1), the 68000 family linker/locator, to form executable programs for the Motorola 68000 family processors.

The *ar68030* and *ar68040* commands are synonyms for the *ar68k* command. They are provide to maintain backward-compatibility with previous versions of these tools.

Individual relocatable files are inserted without change into the archive file. In addition, there is a library symbol table which is used by the linker/locator, *ld68k* (1), to effect multiple passes over the library in an efficient manner.

Individual relocatable files define *modules* which have *modulenames*. The *modulename* is determined in the following way. If the assembly source file contains an **IDNT** directive, then this directive defines the module name. Otherwise, the module name is the name of the assembly source file (with preceding pathname and suffix stripped).

The *ar68k* command operates in either of two modes. The mode is determined by the presence (or absence) of the *archivefile* name.

In the first mode,

ar68k

An *archivefile* is not specified. The *ar68k* command reads librarian commands from standard input. If the standard input is a terminal device, then *ar68k* operates in interactive mode, prompting the user for librarian commands.

The librarian commands are defined below. Additional information may be found in the *HP 68000 Family Assembler/Linker/Librarian User's Guide*. The commands completely control the operation of *ar68k*. The commands specify the name of the archive file and the actions to be performed on the modules which constitute the library.

In the second mode,

```
ar68k [options] [action] ... archivefile
```

all the control information is contained on the command line.

The *archivefile* argument names the archive file to be operated on. If the *archivefile* does not exist, then an empty archive file is created before the actions are performed.

If the archive file name contains a suffix (i.e. contains a period), then the name is used as is to access the archive file. If the archive file name has no suffix, then **.a** is appended to the name before accessing the archive file. In the DOS environment on a PC, the **.lib** suffix is used instead of **.a**.

Action is one of the following:

**-a** *filelist*

The modules contained in the relocatable files in *filelist* are added to the library contained in the archive file. If a module which already exists in the library is added, it is an error.

**-d** *modulelist*

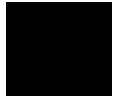
The modules in the *modulelist* are deleted from the library.

**-r** *filelist*

The modules contained in the relocatable files in *filelist* replace modules of the same name in the library.

**-e** *modulelist*

The modules in the *modulelist* are extracted (i.e. copied) and put into relocatable files. The name of the file is the same as the name of the module but with the suffix **.o** appended. In the DOS environment on the PC host, the suffix **.obj** is used instead of **.o**.



Command Syntax  
**ar68k(1)**

In *filelist* (or *modulelist*), individual files (or modules) may be separated by commas or separated by white space with the whole list quoted.

If the file names in file list have a suffix (i.e. contain a period), then the name is used as is to access the relocatable input file. If the name has no suffix, then **.o** (**.obj** on DOS) is appended to the name to obtain the name of the input file.

The following option is recognized by *ar68k*:

**-L** specifies that a library listing file be written to standard output.

LIBRARIAN  
COMMANDS

The *ar68k* command recognizes the following commands. In the syntax descriptions below, square brackets [ ] enclose optional items. Ellipsis ... indicate that the preceding item may be repeated.

**ADDLIB** archivefile [(module [,module] ... )]

Add one or more modules from the named library to the present library. If no modules are specified, the entire library is included.

**ADDMOD** filename [,filename] ...

Add the module contained in one or more relocatable files to the present library.

**CLEAR**

Removes the current library so that another CREATE or OPEN command can be issued.

**CREATE** archivefile

Specify the name of a new archive file to be created.

**DELETE** module [,module] ...

Delete one or more modules from the current library.

**DIRECTORY** archivefile [(module [,module] ...)] [listfile]

Obtain a brief listing of the modules in a library. If no modules are specified, the entire library is listed. If listfile is not specified, the listing goes to standard output.

**END,  
EXIT  
QUIT**

Exit the librarian without saving the current library. Use **SAVE** to save the results of the current session.

**EXTRACT** module [,module] ...

Copy one or more modules to individual relocatable object files. The name of the object file is the module name with **.o** appended. The module name will be appended with **.obj** on DOS machines.

**HELP**

Display the commands (and their syntax) that are valid in the current context.

**LIST** archivefile [(module [,module] ...)] [listfile]

Obtain a detailed listing of the modules in a library. If no modules are specified, the entire library is listed. If **listfile** is not specified, the listing goes to standard output.

**OPEN** archivefile [(module [,module] ...)]

Specify the name of an existing archive file to be opened. If individual modules are specified, only those modules are visible to the librarian while executing subsequent commands. If no modules are specified, all the modules in the existing library are used.

**REPLACE** filename [,filename] ...

Replace one or more existing modules in the present library with the modules from the named files.

**SAVE**

Exit the librarian saving the current library. Use **END** to exit without saving the results of the current session.

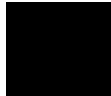
**FILES**

archivefile.a Relocatable archive file. (Unix)

archivefile.lib Relocatable archive file. (DOS)

file.o HP-MRI IEEE-695 relocatable object file. (Unix)

file.obj HP-MRI IEEE-695 relocatable object file. (DOS)

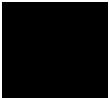


Command Syntax  
**ar68k(1)**

SEE ALSO *HP 68000 Family Assembler/Linker/Librarian User's Guide*, as68k(1), ld68k(1).

DIAGNOSTICS The *ar68k* command returns zero if no errors are detected. It returns non-zero when errors are detected.

Diagnostic messages are displayed on standard error.





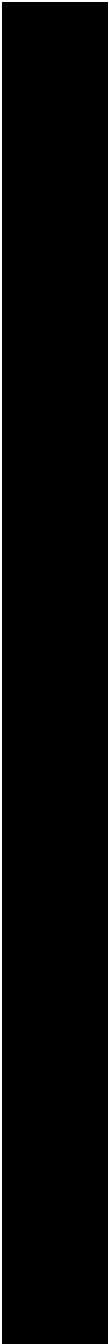
---

## Part 2

---

Reference

**Part 2**





---

## **Assembler Introduction**

This chapter describes the as68k Relocatable Macro Assembler for the 68000 family of microprocessors.

The as68k Relocatable Macro Assembler for the 68000 family of microprocessors translates symbolic machine instructions into binary object code that can be executed by a 68000 family microprocessor. The 68000 family includes the 68000, 68008, 68010, 68302, 68332, and 68020, 68030, and 68040 microprocessors. The instructions specific to the 68881 Floating Point Coprocessor are also translated into the binary code for coprocessor execution.

Object code is produced in a relocatable format by the assembler. Relocatable modules produced by the assembler are linked into a single absolute module by the linking loader.

The as68k mnemonic operation codes, the assembler directives, and the assembler syntax, are all compatible with that used by Motorola in its software products and documentation.

The as68k assembler is a two-pass program that issues helpful error messages, produces an easy to read program listing and symbol table, and outputs a computer readable relocatable object module.

Symbolic information is available for debugging. Assembler symbol files can be produced, and the relocatable object file contains symbolic information which passes through the linker into the IEEE absolute file.

Either the 68000, 68010, 68332, 68020, 68030, or 68040 instruction set may be selected. The assembler will check that only the appropriate instructions are used for the selected processor.

---

## **as68k Features**

Features of as68k include:

- Manufacturer-compatible symbolic machine operation codes (opcodes, directives) are provided.
- Instructions for the 68000 family of microprocessors and the 68881 coprocessor are supported.
- 68030/040 MMU instructions are accepted.
- Conditional assembly is provided.



- User-defined macros are provided.
- Pascal-like run time structured loop control directives are provided.
- Character codes may be specified in ASCII or EBCDIC.
- Case sensitive symbols are supported (with an option to turn off case sensitivity).
- Complex expression evaluation is provided.
- Flexible assembly listing control statements are provided.
- Symbolic or cross reference table listing may be generated.
- Symbols may be included in the output object module for symbolic debugging.
- Relocatable modules may be produced.
- A2-A5 relative addressing is supported.
- Complex relocation is supported in the Loader.
- Supports long file names.

These features aid the program developer in producing well documented, modular, working programs in a minimum of time.

---

## **Assembler Statements**

An assembly language program is comprised of statements written in symbolic machine language. There are four types of assembly language statements:

- Instructions.
- Directives.
- Macros.
- Comments.

All but comment statements are written in the following format:

Label                    Operation                    Operand                    Comment

The various fields that comprise a statement are separated by one or more blanks or tabs, and in some cases, a colon or semicolon. Statements may be a maximum of 512 characters long.

### **Label Field**

The label field assigns a memory address or constant value to the symbolic name contained in the field. The label field may begin in any column if terminated by a colon, or it must begin in column one when the colon is omitted. A label may be the only field in a statement.

The first 31 characters of a label are significant.

Labels are case sensitive by default. You can turn off case sensitivity with the "OPT NOCASE" assembler directive.

### **Operation Field**

The operation field specifies a symbolic operation code, a directive, or a macro call. If present, this field must begin after column one and be separated from the label field by one or more blanks, tabs, or a colon. Assembly language instructions and directives may be upper or lower case. Macros can be case sensitive or not depending on the CASE flag.

### **Operand Field**

The operand field is used to enter arguments for the opcode, directive, or macro specified in the operation field. The operand field, if present, is separated from the operation field by one or more blanks or tabs.

### **Comment Field**

The comment field gives you a place to put messages stating the purpose of a statement or group of statements. The comment field is always optional, and if present, must be separated from the preceding field by one or more blanks, tabs, an exclamation point or a semicolon. For those opcodes and directives



that have optional operands that are not present, the comment field must always start with an exclamation point or a semicolon.

---

## Statement Examples

The next few section give examples of the four types of statements that can be used in assembly language programs.

### Instruction Statement

The instruction statement is a written specification for a particular machine operation, expressed by a symbolic operation code, also called a mnemonic, and operands. Symbolic addresses may be defined by the statement and symbolic addresses may also be used for opcode operands. For example:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
ISAM	MOVE	MEM, D2	

#### Where:

ISAM	A symbol representing the memory address of the instruction.
MOVE	A symbolic opcode representing the bit pattern of the move instruction.
MEM	A symbol representing a memory address.
D2	A reserved symbol representing data register number 2.

## Directive Statement

A directive statement is interpreted as a control statement to the assembler. It is not translated into a machine instruction. For example:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
ABAT	DC	DELT	

### Where:

**ABAT** A symbol. The assembler will assign the value of the location counter to this symbol. The location counter (assembly program counter) contains the address of the first byte of the code generated by the directive DC.

**DC** A directive that instructs the assembler program to allocate two bytes of memory.

**DELT** A symbol representing an address. The address will be placed into the two bytes allocated by the DC directive.

## Macro Statement

A macro statement is a call to a sequence of instructions or a definition of a sequence of instructions as a macro. A call can be made many times from any part of a program as long as the call appears after the macro definition. The chapter "Macros" explains macros in greater detail. The following is an example of a macro definition and call:

```
1          MAC1  MACRO  P1
2          L&&P1  MOVE   D0,D1  ; Create label using parameter.
3          ENDM
4
5          MAC1  XX      ; Call macro.
5.1 00000000 3200  LXX   MOVE   D0,D1  ; Create label using parameter.
6          END
```

## Comment Statement

A comment statement is not processed by the assembler program. Instead, it is reproduced on the assembly listing and may be used to document groups of assembly language statements. A comment statement is indicated by encoding





an asterisk in the first column, or an exclamation point or semicolon as the first nonblank character on a line. For example:

```
* THIS IS A COMMENT STATEMENT  
; THIS IS ALSO A COMMENT STATEMENT
```

Blank lines are also treated as comment statements.

---

## Return Codes

as68k will return 0 if the program assembles without errors. If errors are detected, the assembler will return 1.

Error messages are written to the standard error output and to the assembler listing. Error messages and warnings are listed in the "Assembler Error Messages" appendix.

---

## Assembler Syntax

The assembler language, like other programming languages, has a character set, a vocabulary, rules of grammar, and allows for definition of new words or elements. The rules that describe the language are referred to as the "syntax" of the language.

## Assembler Character Set

The assembler will recognize ASCII characters 20 hex through 7E hex. Any other characters, except in a comment field, will generate an error. Many of the special characters have no predefined meaning except as character constants.

### Alphabetic Characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z  
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

### Numeric Characters:

```
0 1 2 3 4 5 6 7 8 9
```

### Special Characters:

```
      (blank)  
>    (greater than)  
<    (less than)  
'    (single quote)  
,    (comma)  
+    (plus sign)  
-    (minus sign)  
&    (ampersand)  
!    (exclamation)  
"    (double quote)  
#    (sharp)  
%    (percent)  
{    (left curved bracket)  
}    (right curved bracket)  
[    (left square bracket)  
]    (right square bracket)  
^    (up arrow)  
/    (slash)  
$    (dollar)  
*    (asterisk)  
(    (left parenthesis)  
)    (right parenthesis)  
@    (commercial at)  
.    (period)  
:    (colon)  
;    (semi-colon)  
=    (equal sign)  
_    (underbar)  
?    (question mark)  
|    (vertical bar)  
'    (grave accent)  
~    (tilde)  
\    (back slash)
```

---

## Symbols

A symbol is a sequence of characters. The first character in a symbol must be alphabetic or one of the special characters: ? (question mark), . (period), or \_ (underbar). Subsequent characters in the symbol may consist of any of the special characters just mentioned, a \$ (dollar sign), alphabetic letters, or numeric digits. Embedded blanks are not permitted in symbols. Symbols are case sensitive by default. To turn off case sensitivity, use the "OPT NOCASE" assembler directive.

The assembler generates "local" symbols in macros that start with the character sequence \@. However, these symbols are only valid inside a macro.

Symbols may be up to 31 characters in length. They may actually be longer, but only the first 31 characters are used by the assembler for symbol definition.

Symbols are used to represent arithmetic values, memory addresses, bit arrays (masks), etc. Examples of valid symbols are:

```
LAB1
mask
LOOP$NUM
L23456789012345678901234567890123456789 (truncated to 31 characters)
```

Examples of invalid symbols are:

```
ABORT * (contains special character)
1LAR (begins with a numeric)
PAN N (embedded blank, symbol is PAN)
```

Symbols beginning with two or more question marks, for example ??LAB1, are treated slightly differently by the assembler. as68k uses the two question mark convention to identify "assembler generated" symbols. For example, when the assembler creates unique labels in macro expansions, it generates symbols of the form ??0001, ??0002, etc. These assembler generated symbols are not included in the assembler listing or the HP format "asmb\_sym" file unless the OPT G assembler flag is set.

If you code your own symbols beginning with two question marks, these symbols will not be available for debugging unless you specify the OPT G directive.

## Reserved Symbols

The as68k assembler has internally defined the symbolic register names that Motorola uses in their assembly language to denote the various hardware registers. They are:

32-bit address registers	A0, A1, A2, A3, A4, A5, A6, A7, SP
32-bit data registers	D0, D1, D2, D3, D4, D5, D6, D7
control registers	PC, SR, CCR, USP
68331/332/010/030/040 vector base/function code registers	VBR, SFC or SFCR, DFC or DFCR
68020/30/40 cache/stack registers	CACR, CAAR, MSP, ISP
68030/40 MMU registers	CRP, SRP, TC, TT0, TT1, MMUSR, ITT0, ITT1, DTT0, DTT1
68020/30 pseudo registers	ZA0, ZA1, ZA2, ZA3, ZA4, ZA5, ZA6, ZA7, ZD0, ZD1, ZD2, ZD3, ZD4, ZD5, ZD6, ZD7, ZPC
68881 registers	FP0, FP1, FP2, FP3, FP4, FP5, FP6, FP7, FPCR, FPSR, FPIAR, CONTROL, STATUS, IADDR

Users may also define their own keywords with the EQU directive to represent the above predefined registers.

For Example:

```
COUNT    EQU    D4
          ADD.B  #1, COUNT
```

is the same as:

```
ADD.B    #1, D4
```

The reserved symbol "NARG" is used to represent the number of arguments passed on a macro call.



Reserved symbols will not appear in a symbol table or in a cross reference listing.

### Location Counter Symbol (\*)

The asterisk (\*) is the symbol for the "location counter" (also often called the "assembly program counter"). The value of the location counter symbol is the address associated with the first byte of the current instruction. The location program counter symbol can be absolute or relocatable depending on whether it appears in an absolute or relocatable section.

### Symbol Types

The assembler assigns data types to symbols. These data types are transmitted by the assembler and loader to the HP-MRI IEEE-695 absolute file. Debugging tools which consume HP-MRI IEEE-695 files may use these data types when interpreting assembly language modules.

The symbol type is determined by associating the label with an instruction or directive. Instructions are always given the type "Code Address". Directives DC, DS, and DCB have their data types determined by the size extension, as shown in the example below.

The type is determined by the instruction or directive on the same line as the label. If the symbol is defined on a line without an instruction or directive, then the type is determined by the first code generating instruction or directive which follows the label (in the same section). Finally, if no code generating directive follows the unattached label, the label receives type "Code Address".

```

LAB1          ; type Code Address because instruction follows
INST  MOVE   D0,D1 ; type Code Address
LAB2          ; type Unsigned Byte because DC.B directive follows
BYTE  DC.B   0    ; type Unsigned Byte
WORD  DC.W   0    ; type Unsigned Short
LONG  DC.L   0    ; type Unsigned Long
FLOAT DC.S   0.0  ; type 32-Bit Float
DOUBLE DC.D  0.0  ; type 64-Bit Float
XTEND DC.X   0.0  ; type Extended Float
LAB3          ; type Code Address because END follows
END

```

## Constants

A constant is an invariant quantity. It may be an arithmetic value or a character code. Arithmetic values may be represented in either integer or floating point format.

### Integer Constants

In most cases, integer constants must be contained in one, two, or four bytes. A one byte constant can contain an unsigned number with a value from 0 to 255. A two byte unsigned number can range from 0 to 65535. A four byte unsigned number can range from 0 to 4,294,967,295. When a constant is negative, its equivalent two's complement representation is generated and placed in the field specified. A one byte two's complement number may range from -128 to 128. A two byte two's complement negative number may range from -32768 to 32767. A four byte two's complement signed number may range from -2,147,483,648 to 2,147,483,647.

Numbers whose most significant bit is set may be either interpreted as a large positive number or a negative number. For example, the one byte number \$FF may be either + 255 or -1 depending on the usage. The assembler will correctly recognize numbers in either form, but the user is generally responsible for their interpretation.

All constants are evaluated as 32 bit quantities, i.e., modulo  $2^{32}$ . Whenever an attempt is made to place a constant in a field for which it is too large, an error message is generated by the assembler.

Decimal constants may be defined as a sequence of numeric characters optionally preceded by a plus or a minus sign. If unsigned, the value is assumed to be positive.

Constants with bases other than decimal are defined by specifying a coded descriptor or special character before or after the constant. Motorola uses the special characters to indicate base.



The following table lists the available descriptors and their meanings. If no descriptor is given, the number is assumed to be decimal.

**Table 3-1. Constant Base Descriptor Prefixes/Suffixes**

<b>BASE</b>	<b>PREFIX</b>	<b>SUFFIX</b>
<b>Binary</b>	<b>%</b>	<b>B</b>
<b>Octal</b>	<b>@</b>	<b>O, Q</b>
<b>Decimal</b>	<b>none</b>	<b>D</b>
<b>Hexadecimal</b>	<b>\$</b>	<b>H (Leading 0's are required for hex numbers whose first character is not a decimal number.)</b>

Examples of constants are:

```
%1001
@56
640537
$3AB
45
100101B
```

### **Floating-Point Constants**

Floating point numbers may be in either decimal or hexadecimal format. A decimal floating point number must contain either a decimal point or an "E" indicating the beginning of the exponent field. For example: "3.14159", "-22E-100". The latter example means "-22 times (10 to the -100th power)". Underscores may occur before or after the "E" to increase readability. Underscores are ignored in determining the value of a constant.

A hexadecimal floating point number is denoted by a colon ":" followed by a series of hex digits: up to 8 digits for single-precision, 16 digits for double-precision, or 24 digits for extended-precision or packed-decimal. The

**Constants**

digits specified are placed in the field as they stand; the user is responsible for determining how a given floating-point number is encoded in hexadecimal digits. If fewer digits than the maximum permitted are specified, the ones that are present will be *left-justified* within the field. Thus the first digits specified always represent the sign and exponent bits.

Floating-point constants are only permitted in DC, DCB, and FEQU directives.

**Character Constants**

An ASCII or EBCDIC character constant may be specified by enclosing one or more characters within quote marks and preceding them with an A for ASCII or an E for EBCDIC. If no descriptor is specified, the string is assumed to be ASCII. Examples of character constants are:

```
ADD.B    #'Z',D2
EOR      #'0',CCR    ;in hex: F000
ANDI     #'aB',D7
MOVE.L   #'JUMP',(A2)
```

---

**Note**

When character strings are used as operands of word and longword operations, the assembler assigns values according to the following rules. These rules were chosen because they are compatible with the Motorola M68000 Family Resident Structured Assembler.

---





In **DC** directives, character strings are always left justified in words or longwords. Any remaining bytes on the right of the word or longword are filled with zeros. For example:

```
DC.B 'A' ; Hex value is 41
DC.B 'AB' ; Hex value is 41 42
DC.W 'A' ; Hex value is 4100
DC.W 'AB' ; Hex value is 4142
DC.W 'ABC' ; Hex value is 4142 4300
DC.L 'A' ; Hex value is 41000000
DC.L 'AB' ; Hex value is 41420000
DC.L 'ABC' ; Hex value is 41424300
DC.L 'ABCD' ; Hex value is 41424344
DC.L 'ABCDE' ; Hex value is 41424344 45000000
```

In any other context, the justification depends on the number of characters in the string. Strings that are 1 or 2 characters long are left justified to the nearest word boundary. Strings that are 3 or 4 characters long are left justified in the longword. Remaining bytes on the right are zero filled. For example:

```
MOVE.B #'A',D0 ; Value moved is hex 41
MOVE.W #'A',D0 ; Value moved is hex 4100
MOVE.W #'AB',D0 ; Value moved is hex 4142
MOVE.L #'A',D0 ; Value moved is hex 00004100 NOTE!
MOVE.L #'AB',D0 ; Value moved is hex 00004142 NOTE!
MOVE.L #'ABC',D0 ; Value moved is hex 41424300 NOTE!
MOVE.L #'ABCD',D0 ; Value moved is hex 41424344 NOTE!
```

To generate code for a single quotation mark (or a caret) in a character constant or string delimited by single quotes (or carets), it must be specified as two single quote marks (or two carets). For example:

```
'DON''T'
^THE ' AND ^^ DELIMITERS^
```

The code for a single quote mark will be generated once for every two quote marks that appear contiguously within the character string.

## Expressions

An expression is a sequence of one or more symbols, constants or other syntactic structures separated by arithmetic operators. Expressions are evaluated left to right, subject to the precedence of operators shown below. Parentheses may be used to establish the correct order of the arithmetic operators and it is recommended that they be used in complex expressions involving operators such as >>, &, =, etc. The following table summarizes the operators and their precedences:

**Table 3-2. Operator Precedence**

Precedence	Operator	
1	==	test for existing operand
2	+	unary plus
	-	unary minus
	"	logical NOT
	.SIZEOF.	size of combined section
	.STARTOF.	starting address of combined section
3	>>	shift right
	<<	shift left
4	&	logical AND
	!	logical OR
	!!	exclusive OR
5	*	multiplication
	/	division
6	+	addition
	-	subtraction

Precedence	Operator (Cont'd)	
7	= , < >	equality, not equality
	> , > =	greater than, greater or equal
	< , < =	less than, less or equal

The == operator is used to determine whether an operand exists. This is further described in the "Macro Call" section of the "Macros" chapter.

The .STARTOF. and .SIZEOF. operators help you to write code that initializes or copies logical sections of memory. The section being referenced in these operators must have been previously defined in a SECT or COMMON directives.

The .STARTOF. operator gives the starting address of the combined section in which the named subsection will be contained. The .SIZEOF. operator gives the size of the combined section.

The comparison operators = , > = , etc., return a logical True (all one bits) if the comparison is true and a logical False (zero) if the comparison is not true. All operands are considered to be unsigned 32 bit values and the comparison is unsigned. (Thus, comparisons against 0 in particular are not very useful.) An example follows:

```
IF      DATA=5
```

The shift operators (>> , <<) shift the argument that goes before the operator right or left the number of bits specified by the argument that follows the operator. Zeros are shifted into the high or low order bits. An example follows:

```
DC .B   2<<BIT
```

---

**Note**

Embedded blanks are not allowed in expressions. The assembler interprets spaces as termination characters. Expressions are limited to about 45 separate "entities" per expression. An entity could be a symbol, an operator, a literal, parentheses, and so on. If you find that you must have an expression with an over-limit number of entities, you may be able to use EQUs to break up the expression into subexpressions and not exceed the limit.

---

All expressions are evaluated modulo  $2^{32}$  and must resolve to a single unique value that can be contained in 32 bits. Consequently, character strings longer than four characters are not permitted in expressions. When an attempt is made to place an expression in a one or two byte field and the calculated result is too large to fit, an error message is generated. Examples of valid expressions:

```
PAM+3  
LOOP+( ADDR>>8 )  
( PAM+$45 ) /CAL  
VAL,1=VAL,2  
IDAM&255
```

---

## Assembler Listing Description

As previously stated, the as68k assembler uses two passes. During the first pass, macros are expanded, labels are examined and placed into the symbol table, opcodes and directives are decoded, and statement byte lengths are determined so the location counter may be updated.

During the second pass, the object code is generated, symbolic addresses are resolved, and a listing and output object module are produced. Errors detected during the assembly process will be displayed on the output listing with a cumulative error count also given.

At the end of the assembly process a symbol table or a cross reference table may be displayed.

### Assembler Listing

During pass two of the assembly process, a program listing is produced. The main purpose of the listing is to convey all pertinent information about the assembled program, the memory addresses, and their values. The load module, also produced during pass two, contains the object code address and value information, but in a format that is easily read by computers.

The following points may help you better understand the listing format.



- When the assembler detects error conditions during the assembly process, an "ERROR" message will appear on the line following the source code which caused the error. An explanation of the individual assembler warnings and errors is given in the "Assembler Error Messages" appendix.
- The column titled "Line" contains decimal numbers that are associated with the listing source lines. These numbers are referred to in the cross reference table. The numbers can include periods (.) separating the digits. These periods provide a distinction between nesting levels of included or macro expanded code.
- The column titled "Address" contains a value that represents the first memory address of any object code generated by this statement or the value of an EQU or SET or FAIL directive.
- To the right of the address are up to three words of object code generated by the assembly language source statement. Additional words of object code are shown on subsequent lines. The first hexadecimal number represents one word of data to be stored in the memory address and the memory address plus one. If there are additional words, they will be stored in subsequent memory locations.
- To the right of the data words are the assembler relocation flags. The flags are:
  - R - relocatable operand.
  - E - external operand.
  - C - complex relocatable operand.If one operand is relocatable and another external, an E will be displayed.
- The user's original source statements are reproduced to the right of the above information.
- At the end of the listing the assembler prints the message "Errors: nnnnn, Warnings: nnnnn". Warnings are marked by a WARNING message; errors are marked by an ERROR message. See the "Assembler Error Messages" appendix for a complete list of error messages. The assembler substitutes two words of NOP's when it cannot translate a particular opcode and so provides room for patching the program.
- A symbol table or cross reference table is generated at the end of the assembly listing. The table lists all symbols defined in alphabetical order, along with the section in which they were defined, as well as their final absolute values. Line numbers in which the labels occur are listed under "REFERENCES".

### **Cross Reference Table Format**

The cross reference option is turned off by default. To turn it on, use "OPT X"; and to turn it off again, use "OPT -X" (see the OPT description in the "Assembler Directives" chapter). The assembler will produce a symbol table, and the symbol table will contain cross reference information if "OPT X" has been specified.

You can limit the listing of cross references to selected portions of the program by turning the cross reference option on and off. However, to obtain the cross reference listing, the option must be turned on before the END directive. Typically, the "OPT X" directive will be one of the first statements in the source program and will never be turned off.

All symbols defined by the user in the program are listed under the heading "LABEL". The symbol values are listed under "VALUE". Any flag to the left of the values indicates the relocation type of the symbols.

Under REFERENCES, a line number preceded by a minus sign indicates that the symbol was defined on that line. Line numbers not preceded by a minus sign indicate a reference to a symbol. If no line numbers appear, the symbol is the internal system symbol NARG. Note that for SET symbols or for multiply defined symbols, more than one definition may appear for the symbol. Section names, macro names, and the module name do not appear in the symbol table listing.

---

# 4



---

## Instructions and Address Modes

This chapter describes the instructions and address modes used by the 68000 family and 68881 processors.

## Instructions and Address Modes

This chapter describes:

- The 68000 family and 68881 assembly language instruction mnemonics and qualifiers.
- How the assembler will generate code for variants of certain instructions depending on the instruction's operands.
- The address modes for the 68000 family microprocessors.
- Assembler syntax and the address modes which are generated for a particular syntax.
- How you can control the generation of address modes by setting or clearing various assembler options (with the OPT directive).



## Instructions

The assembler instructions and their legal operands are defined in the following Motorola publications:

- *MC68000 16/32-bit Microprocessor User's Manual* (Fourth Edition MC68000UM(AD4))
- *MC68020 32-bit Microprocessor User's Manual* (Second Edition MC68020UM/AD)
- *MC68030 Enhanced 32-bit Microprocessor User's Manual* (MC68030UM/AD)
- *MC68040 32-bit Microprocessor User's Manual* (MC68040UM/AD)
- *CPU32 32-Bit Instruction Processor Reference Manual* (Preliminary Rev. 0.8)
- *MC68881 Floating-Point Coprocessor User's Manual* (MC68881UM/AD)
- *M68000 Family Resident Structured Assembler Reference Manual* (M68KMASM/D10)

Sometimes, the Motorola assembler manual and the Motorola processor manuals define different mnemonics for the same operation. as68k generally recognizes both methods.



## Instructions and Address Modes

### Qualifiers

---

#### Caution

---

The following instructions do not act as you might expect.

```
DIVS.L a,Dq ;Dq is both upper & lower half of 64 bit dividend.  
DIVU.L a,Dq ;Dq is both upper & lower half of 64 bit dividend.
```

These instructions divide a 64 bit dividend by a 32 bit divisor and put a 32 bit quotient into Dq. The 64 bit dividend is formed by using Dq as both the high half and low half of the number. This is not a very useful operation.

The assembler's behavior contradicts the description in Motorola's *MC68020 32-bit Microprocessor User's Manual*. However, the behavior is compatible with the Motorola *M68000 Family Resident Structured Assembler* and was chosen for that reason.

In order to divide a 32 bit dividend and obtain a 32 bit quotient, write the following.

```
DIVSL.L a,Dq ; 32/32 == 32q  
TDIVS.L a,Dq ; 32/32 == 32q  
DIVUL.L a,Dq ; 32/32 == 32q  
TDIVU.L a,Dq ; 32/32 == 32q
```

In order to divide a 64 bit dividend in a sensible way, write the following.

```
DIVS.L a,Dr:Dq ; 64/32 == 32q,32r
```

---

## Qualifiers

Instruction mnemonics may in some cases have a qualifier (also called an extension), which is separated from the mnemonic by a period.

### Scope Qualifiers

The qualifier field usually is used to specify the scope of operation for an instruction. For this purpose, the recognized codes are ".B" (byte), ".W" (word), and ".L" (longword). If an instruction which may have more than one qualifier is coded without one, ".W" is the default.

A few instructions use the qualifier field to force the assembler to override its defaults in choosing the short or long form of an instruction; the recognized codes in this case are ".S" (short) and ".L" (long).

### **Floating Point Qualifiers**

Floating point operations use the ".W", ".B", and ".L" integer qualifiers as well as additional qualifiers for real numbers. The floating point qualifiers are ".S" for single precision real, ".D" for double precision real, ".X" for extended precision real, and ".P" for packed decimal string real.



---

### **Mnemonics**

A list of the allowable 68000 family instruction mnemonics is shown in table 2-1. The legal qualifiers for each are listed. If no qualifiers are listed after a mnemonic, none are legal. Footnotes are used to provide additional information.

The notation "cc" (lower case) indicates one of the condition codes: T, F, HI, LS, CC (or HS), CS (or LO), NE, EQ, VC, VS, PL, MI, GE, LT, GT or LE.

The processor and FPU instructions shown in the tables use this notation for the qualifiers:

<b>Qualifier</b>	<b>Meaning</b>
B, W, L	68000 sizes; specifies signed integer data types of byte (8 bits), word (16 bits), or long word (32 bits).
S	Single precision real (32 bits).
D	Double precision real (64 bits).
X	Extended precision real (96 bits).
P	Packed Decimal (BCD) string real (96 bits, 12 bytes).
FPn	One of the 8 floating point data registers.
FPcr	One of the 3 floating point system control registers (control - FPCR, status - FPSR, or iaddr - FPIAR).
cc	Index into the 68881 constant ROM.

Instructions and Address Modes  
**Mnemonics**

**Table 4-3. Instruction Mnemonics**

(note)	Mnemonic	Qualifiers	
	ABCD	B	
	ADD	B W L	
	ADDA	W L	
	ADDI	B W L	
	ADDQ	B W L	
	ADDX	B W L	
	AND	B W L	
	ANDI to CCR	B	
	ANDI to SR	W	
	ANDI to other	B W L	
	ASL	B W L	
	ASR	B W L	
(6)	Bcc	B W L S	(BT and BF are invalid - use BRA)
(7)	BCHG	B L	
(7)	BCLR	B L	
(1)	BFCHG		
(1)	BFCLR		
(1)	BFEXTS		
(1)	BFEXTU		
(1)	BFFFO		
(1)	BFINS		
(1)	BFSET		
(1)	BFTST		
(8)	BGND		
(2)	BKPT		
(6)	BRA	B W L S	
(7)	BSET	B L	
(6)	BSR	B W L S	
(7)	BTST	B L	
(1)	CALLM		
(1)	CAS	B W L	
(1)	CAS2	B W L	
(3)	CHK	W L	
(10)	CHK2	B W L	
(11)	CINVA		
(11)	CINVL		
(11)	CIVNP		
	CLR	B W L	
	CMP	B W L	
	CMPA	W L	
	CMPI	B W L	
	CMPM	B W L	
(10)	CMP2	B W L	
(11)	CPUSHA		
(11)	CPUSHL		
(11)	CPUSHP		
	DBcc	W	(DBRA is also legal; same as DBF)

Table 4-1. Instruction Mnemonics (Cont'd)

(note)	Mnemonic	Qualifiers	
(3)	DIVS	W L	(with .L, 64-bit dividend/32-bit divisor)
(10)	DIVSL	L	(32-bit dividend/32-bit divisor)
(3)	DIVU	W L	(with .L, 64-bit dividend/32-bit divisor)
(10)	DIVUL	L	(32-bit dividend/32-bit divisor)
	EOR	B W L	
	EORI to SR	W	
	EORI to CCR	B	
	EORI to other	B W L	
	EXG	L	
(4)	EXT	W L	
(4)	EXTB	W L	
(4)	EXTW	L	
	ILLEGAL		
	JMP		
	JSR		
	LEA	L	
(3)	LINK	W L	
(8)	LPSTOP		
	LsL	B W L	
	LsR	B W L	
(2)	MOVE from CCR	W	
	MOVE to CCR	W	
	MOVE from SR	W	
	MOVE to SR	W	
	MOVE to/from USP	L	
	MOVE other	B W L	
	MOVEA	W L	
(2)	MOVEC	L	
	MOVEM	W L	
	MOVEP	W L	
	MOVEQ	L	
(2)	MOVES	B W L	
(11)	MOVE16	L	
(3)	MULS	W L	
(3)	MULU	W L	
	NBCD	B	
	NEG	B W L	
	NEGX	B W L	
	NOP		
	NOT	B W L	
	OR	B W L	
	ORI to CCR	B	
	ORI to SR	W	
	ORI to other	B W L	
(1)	PACK		
	PEA	L	

Instructions and Address Modes  
**Mnemonics**

**Table 4-1. Instruction Mnemonics (Cont'd)**

(note)	Mnemonic	Qualifiers	
	(11) PFLUSHAN		
(11)	PFLUSHN		
(10)(11)	PLOAD		
(10)(11)	PMOVE		
(11)	PTEST		
	RESET		
	ROL	B W L	
	ROR	B W L	
	ROXL	B W L	
	ROXR	B W L	
(2)	RTD		
	RTE		
(1)	RTM		
	RTR		
	RTS		
	SBCD	B	
	Scc	B	
	STOP		
	SUB	B W L	
	SUBA	W L	
	SUBI	B W L	
	SUBQ	B W L	
	SUBX	B W L	
	SWAP	W	
	TAS	B	
(8)	TBLS	B W L	
(8)	TBLSN	B W L	
(8)	TBLU	B W L	
(8)	TBLUN	B W L	
(10)	TDIVS	L	(same as DIVSL, 32-bit dividend/32-bit divisor)
(10)	TDIVU	L	(same as DIVUL, 32-bit dividend/32-bit divisor)
	TRAP		
(5)	TRAPcc	W L	
(5)	Tcc		
(5)	TPcc	W L	
	TRAPV		
	TST	B W L	
	UNLK		
(1)	UNPK		

**Table 4-1. Instruction Mnemonics (Cont'd)**

NOTES:

- (1): 68020/68030 or CPU32 only.
- (2): 68010 or greater only.
- (3): The .L qualifier is valid only for 68331/332, 68020 and greater processors. Cannot be used in code that will target chips less capable than the 68331/332.
- (4): There are 3 distinct Extend operations. Extend Byte to Word may be coded as EXT.W or EXTB.W. Extend Word to Long may be coded as EXT.L or EXTW.L. Extend Byte to Long, which is valid only for 68331/332, 68020 and greater processors, must be coded as EXTB.L.
- (5): TRAPcc, Tcc and TPcc (68331/332, 68020 and greater processors) are different mnemonics for the same instructions. TRAPcc may or may not take an operand; Tcc may not have an operand, and TPcc must have an operand.
- (6): The ".B" extension forces a Byte instruction. The ".W" extension forces a Word instruction. The ".L" extension forces a Word instruction when a chip other than the 68331/332 or 68020 is targeted. When the 68331/332, 68020 or greater processor is targeted, a Longword instruction is forced unless the OPT OLD directive is used to force Word instruction to be used.
- (7): For the single-bit instructions, the generated code is fully determinable from the operands and therefore the qualifier serves no function. For compatibility, however, the qualifiers .B and .L are accepted, and if a qualifier is present the operands are checked to be sure they match the qualifier.
- (8): 68331 and 68332 only. Cannot be used in any other target.
- (9): 68030 MMU instructions. These instructions have several variations.
- (10): Cannot be used for target less capable than 68331/332.
- (11): Added or modified for 68040.
- (12): 68030 only.



## Floating Point Mnemonics

A list of the allowable instruction mnemonics for the 68881 floating point coprocessor and the 68040 floating point unit is shown in table 2-2. The legal qualifiers for each are listed. If no qualifiers are listed after a mnemonic, none are legal. Footnotes are used to provide additional information.

The notation "cc" (lower case) indicates one of the condition codes: GT, GE, LT, GL, LE, GLE, SEQ, ST, NGT, NGE, NLT, NGL, NLE, NGLE, SNEQ, SF, OGT, OGE, OLT, OGL, OLE, OR, EQ, T, ULE, ULT, UGE, UEQ, UGT, UN, NEQ, or F.

An "F" in the 68881 column means that the instruction is supported by the 68881/882. An "F" in the 68040 column means that the instruction is fully supported in the 68040 hardware. A "P" in the 68040 column means that the instruction is supported in hardware except for the packed decimal formats.

Unimplemented 68881/882 instructions are trapped by the 68040. The instructions may then be handled using software routines.



**Table 4-2. 68881 Instruction Mnemonics**

(note)	Mnemonic	68881	68040	Qualifiers
(3)	FABS	F	P	B W L S D X P
(3)	FACOS	F		B W L S D X P
(2)	FADD	F	P	B W L S D X P
(3)	FASIN	F		B W L S D X P
(3)	FATAN	F		B W L S D X P
(3)	FATANH	F		B W L S D X P
(4)	FBcc	F	F	W L
(2)	FCMP	F	P	B W L S D X P
(3)	FCOS	F		B W L S D X P
(3)	FCOSH	F		B W L S D X P
(4)	FDBcc	F	F	W
	FDABS		F	B W L S D X
	FDADD		F	B W L S D X
	FDDIV		F	B W L S D X
(2)	FDIV	F	P	B W L S D X P
	FDMOVE		F	B W L S D X
	FDMUL		F	B W L S D X
	FDNEG		F	B W L S D X
	FDSQRT		F	B W L S D X
	FDSUB		F	B W L S D X
(3)	FETOX	F		B W L S D X P
(3)	FETOXM1	F		B W L S D X P
(3)	FGETEXP	F		B W L S D X P
(3)	FGETMAN	F		B W L S D X P
(3)	FINT	F		B W L S D X P
(3)	FINTRZ	F		B W L S D X P
(3)	FLOG10	F		B W L S D X P
(3)	FLOG2	F		B W L S D X P
(3)	FLOGN	F		B W L S D X P
(3)	FLOGNP1	F		B W L S D X P
(2)	FMOD	F		B W L S D X P
(1)	FMOVE to FPn	F	P	B W L S D X P
(1)	FMOVE from FPn	F		B W L S D X P
(1)	FMOVE FPcr	F		L
(1)	FMOVECR	F		B W L S D X P
(1)	FMOVEM FPn	F	F	L X
(1)	FMOVEM FPcr	F	F	L X
(2)	FMUL	F	P	B W L S D X P
(3)	FNEG	F	P	B W L S D X P
(4)	FNOP	F	P	
(2)	FREM	F		B W L S D X P
(5)	FRESTORE	F	F	
	FSABS		F	B W L S D X
	FSADD		F	B W L S D X
(5)	FSAVE	F	F	
(2)	FSCALE	F		B W L S D X P
(4)	FScC	F	F	B

Instructions and Address Modes  
**Floating Point Mnemonics**

**Table 4-2. 68881 Instruction Mnemonics (Cont'd)**

(note)	Mnemonic	68881	68040	Qualifiers
	FSDIV		F	B W L S D X
(2)	FSGLDIV	F		B W L S D X P
(2)	FSGLMUL	F		B W L S D X P
(3)	FSIN	F		B W L S D X P
(3)	FSINCOS	F		B W L S D X P (dual monadic)
(3)	FSINH	F		B W L S D X P
	FSMOVE		F	B W L S D X
	FSMUL		F	B W L S D X
	FSNEG		F	B W L S D X
(3)	FSQRT	F	P	B W L S D X P
	FSSQRT		F	B W L S D X
	FSSUB		F	B W L S D X
(2)	FSUB	F	P	B W L S D X P
(3)	FTAN	F		B W L S D X P
(3)	FTANH	F		B W L S D X P
(3)	FTENTOX	F		B W L S D X P
(5)	FTRAPcc	F	F	W L
(4)	FTcc	F	F	
(4)	FTPcc	F	F	W L
(4)	FTST	F	P	B W L S D X P
(3)	FTWOTOX	F		B W L S D X P

NOTES:

(1): 68881 Data Movement instruction. Moves operands into, between, or out of the floating point data registers.

(2): 68881 Dyadic Operation instruction. Performs arithmetic operations requiring two operands, e.g. subtract. One of the operands is always from a floating point data register; the other may be from a memory address register, from an integer data register, or from a floating point data register. The result is stored in a floating point data register.

(3): 68881 Monadic Operation instruction. Performs arithmetic operations that require only one operand, e.g. cosine. The operation is performed on the source operand; the result is stored in a destination which is always a floating point data register that you must specify.

(4): 68881 Program Control instruction. Tests an operand instruction for condition codes set in a floating point status register. The branch instructions within this group allow the user to set a variable based on the floating point condition codes; then use this variable in other program and system control instructions.

(5): 68881 System Control instruction. Communicates with the operating system using a conditional trap instruction. This type of instruction utilizes the same conditional tests as the program control instruction and additionally allows a 16- or 32-bit operand into the instruction for the purpose of passing information to the operating system.

---

## Variants of Instruction Types

The assembler allows you to use "generic" instruction types when writing your programs, and it will generate code for variants of the instruction where appropriate. The assembler generates code for variants of an instruction either because the variant form is implied by the operands or because fewer bytes of code are generated for the variant instruction.

The variants recognized by the assembler are:

Generic	Variants
ADD	ADD, ADDA, ADDQ, ADDI, ADDX
AND	AND, ANDI
CMP	CMP, CMPA, CMPM, CMPI, CMP2
EOR	EOR, EORI
MOVE	MOVE, MOVEA, MOVEQ, MOVEM, MOVEP, MOVES
OR	OR, ORI
SUB	SUB, SUBA, SUBQ, SUBI, SUBX

Example:

```
D250    ADD (A0),D1
D2D0    ADD (A0),A1    ; ADDA
5E50    ADD #7,(A0)   ; ADDQ
0650 FFFF    ADD #$ffff,(a0) ; ADDI
```

When the ADD and SUB instructions have operands which are legal for either the ADDQ or the ADDI variant (for example, # 1,D4), the assembler chooses ADDQ or SUBQ because these instructions are two bytes shorter than ADDI. You can, however, force the ADDI form by specifying the ADDI mnemonic.

We recommend that you use the mnemonics of the variant forms because the resulting code is easier to understand.

## Instruction Operands

In general, instructions have zero, one, two or three operands, and in some cases the same mnemonic may take different numbers of operands to indicate different functions. Not all address modes are necessarily legal for a particular operand of a particular instruction. The legal address modes for an operand vary in an irregular way, which is fully described in the *Motorola 32-Bit Microprocessor User's Manual* (68020/30/40), the *Motorola Floating-Point Coprocessor User's Manual* (68881), and *16/32-bit Microprocessor Programmer's Reference Manual* (other 68000 family processors). There are differences in legal address modes between chips, which are described in detail in these Motorola manuals.

---

## Registers

The assembler recognizes the register mnemonics listed and described below. Register mnemonics may be upper or lower case, and are reserved symbols.

### Data Registers

D0-D7	32-bit Data Registers.
ZD0-ZD7	Suppressed Data Registers (68020/30 only). The register specified is used in the instruction, but its value is taken to be zero for effective address calculations.

### Address Registers

A0-A7	32-Bit Address Registers.
ZA0-ZA7	Suppressed Address Registers (68020/30 only). The register specified is used in the instruction, but its value is taken to be zero for effective address calculations.

### Stack Registers

A7, SP	System Stack Pointer.
--------	-----------------------

USP	User Stack Pointer (for user state).
MSP	Master Stack Pointer (68020 supervisor state).
ISP	Interrupt Stack Pointer (68020 interrupt state).

**Status Registers**

CCR	Condition Code Register. The CCR is the lower eight bits of the status register (SR).
SR	Status Register. All 16 bits can be modified in the supervisor state; only the lower 8 (CCR) can be modified in the user state. (Note that STATUS is the name for the <i>floating-point</i> status register.)
MMUSR	MMU Status Register (68040 only). Contains memory management status information.

**Program Counter Registers**

PC	Program Counter (used in PC relative address modes). The program counter contains the address of the location two bytes beyond the beginning of the currently executing instruction. The user mnemonic PC does not directly access the program counter register, but is used to force the use of program counter relative address modes.
ZPC	Suppressed Program Counter (68020/30 only). The PC is used in the instruction, but its value is taken to be zero for effective address calculations.

**Function Code Registers**

SFC, SFCR	Alternate Function Code Source Register (68010/20/30/40 only).
DFC, DFCR	Alternate Function Code Destination Register (68010/20/30/40 only).



## Instructions and Address Modes Registers

### Cache Registers (68020/30/40 only)

CACR	Cache Control Register. Controls on-chip instruction and data caches.
CAAR	Cache Control Register (68020/30). Holds the address for cache control functions.

### Root Pointer Registers (68030/40 only)

CRP	CPU Root Pointer. Points to root of translation tree for currently executing task.
SRP	Supervisor Root Pointer. Points to root of translation tree that describes supervisor address space.
URP	User Root Pointer. Points to root of translation tree that describes user address space.

### Translation Registers (68030/40 only)

TC	Translation Control register. Controls address translation.
TT0, TT1	Transparent Translation registers. Each specifies separate blocks of memory that are directly addressable without address translation. (68030 only)
ITT0, ITT1	Instruction Transparent Translation registers. Each specifies separate blocks of instruction memory that are directly addressable without address translation. (68040 only)
DTT0, DTT1	Instruction Transparent Translation registers. Each specifies separate blocks of data memory that are directly addressable without address translation. (68040 only)

**Floating Point Registers**

FP0-FP7	Floating-Point Data Registers (68881 and 68040).
FPCR, CONTROL	Floating-Point Control Register (68881 and 68040).
FPSR, STATUS	Floating-Point Status Register (68881 and 68040).
IADDR/FPIAR	Floating-Point Instruction Address Register (68881 and 68040).



**Other Registers**

VBR	Vector Base Register (68010/20/30/40). Used for multiple vector table areas.
-----	--

The 68881 floating point coprocessor uses the 68020 instruction set and addressing modes to provide a logical extension to the integer capabilities of the 68020 processor. In addition to the eight 32-bit Address Registers (A0 to A7), and eight 32-bit Integer Data Registers (D0 to D7) of the 68020, the 68020/68881 processor combination provides eight Floating Point Data Registers (FP0 to FP7). The 68881 interfaces to the 68020 transparently. You access the floating point registers of the 68881 as though they were resident in the 68020. The 68881 coprocessor interface places no restrictions on the use of the 68020 registers. Floating point operations are coded exactly the same as integer operations.

## Address Modes

The Motorola 68000/HC001/08/10/302 supports a basic set of addressing modes. For the purposes of representation and explanation, we can refer to the 68000 addressing modes as the “68000 model.” The Motorola 68020 supports, in addition to the basic 68000 model modes, additional addressing modes and expanded functionality for some of the basic 68000 modes. The “68020 model” is a superset of the 68000 model. The Motorola 68331/332 supports all the basic 68000 modes and a some of the additional modes of the 68020. The “68332 model” is a superset of the 68000 model and a subset of the 68020 model. The Motorola 68030/40 supports the 68000, the 68332, and the 68020 addressing models. The following table summarizes the addressing models supported by each microprocessor:

**Table 4-3. Address Models**

<b>Processor</b>	<b>Address Model Supported</b>
68000	68000 Model
68HC001	68000 Model
68008	68000 Model
68010	68000 Model
68302	68000 Model
68331	68000 Model 68332 Model
68332	68000 Model 68332 Model
68020	68000 Model 68332 Model 68020 Model
68030	68000 Model 68020 Model
68040	68000 Model 68020 Model



Understanding the differences among the addressing models is important for two reasons:

- Incompatibilities and errors can occur if you choose addressing modes from a model not supported by your target processor or that conflict with `CHIP` or `OPT P=` directives.

For instance, if you specify a processor that supports the 68000 model (68000 or 68010) with a `CHIP` or `OPT P=` directive and then use instructions that use 68020 model addressing modes, the assembler will error. Or, if you choose 68020 model addressing modes and a compatible `CHIP` or `OPT P=` directive, problems may still occur if you attempt to execute the code on a processor that supports *only the 68000 model modes*.

- Incompatibilities and errors can also occur if the as68k chooses addressing modes (based on the manner in which operands were specified) from a model not supported by your target processor or that conflict with `CHIP` or `OPT P=` directives.



## The 68000 Model

The 68000 model defines twelve addressing modes. These modes are valid for all 68000 family processors. User's manuals for older chips (68000, for instance) group these addressing modes in broad terms. We choose, however, to define them more explicitly. Table 2-4 summarizes the addressing modes common to all 68000 family chips. (Each addressing mode is preceded by a roman numeral. These roman numerals will be used through the rest of the manual as a short form for these addressing modes.)

**Table 4-4. 68000 Model Addressing Modes**

<p><b>Register Direct Modes</b></p> <ul style="list-style-type: none"><li>I) Data Register Direct</li><li>II) Address Register Direct</li></ul> <p><b>Register Indirect Modes</b></p> <ul style="list-style-type: none"><li>III) Address Register Indirect</li><li>IV) Address Register Indirect with Postincrement</li><li>V) Address Register Indirect with Predecrement</li><li>VI) Address Register Indirect with (16-bit) Displacement</li></ul> <p><b>Register Indirect with Index Modes</b></p> <ul style="list-style-type: none"><li>VII) Address Register Indirect with (8-bit) Displacement and Index</li></ul> <p><b>Absolute Address Modes</b></p> <ul style="list-style-type: none"><li>VIII) Absolute Short Address</li><li>IX) Absolute Long Address</li></ul> <p><b>Program Counter Indirect with Displacement Mode</b></p> <ul style="list-style-type: none"><li>X) Program Counter Indirect with (16-bit) Displacement</li></ul> <p><b>Program Counter Indirect with Index Modes</b></p> <ul style="list-style-type: none"><li>XI) Program Counter Indirect with (8-bit) Displacement and Index</li></ul> <p><b>Immediate Data</b></p> <ul style="list-style-type: none"><li>XII) Immediate</li></ul>
---

## The 68020 Model

The expanded addressing modes for the 68020 model are variations of two of the 68000 model modes. They are the **Address Register Indirect with (8 bit) Displacement and Index** and **Program Counter Indirect with (8 bit) Displacement and Index**. In the 68000 model, these two modes have a specially formatted word of extension not found in the other ten modes. In the 68020 model, these two modes also have a specially formatted word of extension. The interpretation of that extension word can be slightly different, however, in the 68020 model. The six variations defined for these two modes also have a specially formatted extension word and may be followed by additional words of extension. These differences between the two modes give the 68020 model much expanded capabilities over the 68000 model. Table 2-6 summarizes the variations and additions of the 68020 model. (The subscripted roman numerals will be used to refer to these 68020 model modes later in the manual.)

**Table 4-5. 68020 Model Varied and Additional Modes**

<b>Register Indirect with Index Modes</b>	
VIIa)	Address Register Indirect with (8-bit) Displacement and Index *
VIIb)	Address Register Indirect with (16- or 32-bit) Base Displacement and Index
<b>Memory Indirect Address Modes</b>	
VIIc)	Memory Indirect Post-Indexed
VIId)	Memory Indirect Pre-Indexed
<b>Program Counter Indirect with Index Modes</b>	
XIa)	Program Counter Indirect with (8-bit) Displacement and Index *
XIb)	Program Counter Indirect with (16- or 32-bit) Base Displacement and Index
<b>Program Counter Memory Indirect Modes</b>	
XIc)	Program Counter Memory Indirect Post-Indexed
XId)	Program Counter Memory Indirect Pre-Indexed
* In these modes, you may specify a scale factor of 2, 4, or 8. The 68000 model only allows a scale factor of 1.	

## The 68332 Model

The 68332 model is a superset of the 68000 model and a subset of the 68020 model. Table 2-5 summarizes the addressing modes. Each addressing mode is preceded by a roman numeral. These roman numerals will be used through the rest of the manual as a short form for these addressing modes. Roman numerals VIIa, VIIb, XIa, and XIb correspond to the addressing modes found in the 68020 model (table 2-6). All addressing modes except VIIb and XIb are also found in the 68000 model.

**Table 4-6. 68332 Model Addressing Modes**

<p><b>Register Direct Modes</b></p> <ul style="list-style-type: none"><li>I) Data Register Direct</li><li>II) Address Register Direct</li></ul> <p><b>Register Indirect Modes</b></p> <ul style="list-style-type: none"><li>III) Address Register Indirect</li><li>IV) Address Register Indirect with Postincrement</li><li>V) Address Register Indirect with Predecrement</li><li>VI) Address Register Indirect with (16-bit) Displacement</li></ul> <p><b>Register Indirect with Index Modes</b></p> <ul style="list-style-type: none"><li>VIIa) Address Register Indirect with (8-bit) Displacement and Index *</li><li>VIIb) Address Register Indirect with (16- or 32-bit) Base Displacement and Index</li></ul> <p><b>Absolute Address Modes</b></p> <ul style="list-style-type: none"><li>VIII) Absolute Short Address</li><li>IX) Absolute Long Address</li></ul> <p><b>Program Counter Indirect with Displacement Mode</b></p> <ul style="list-style-type: none"><li>X) Program Counter Indirect with (16-bit) Displacement</li></ul> <p><b>Program Counter Indirect with Index Modes</b></p> <ul style="list-style-type: none"><li>XIa) Program Counter Indirect with (8-bit) Displacement and Index *</li><li>XIb) Program Counter Indirect with (16- or 32-bit) Base Displacement and Index</li></ul> <p><b>Immediate Data</b></p> <ul style="list-style-type: none"><li>XII) Immediate</li></ul> <p>* In these modes, you may specify a scale factor of 2, 4, or 8. The 68000 model only allows a scale factor of 1.</p>
---

## **Explanations of Address Modes**

The Program Counter relative modes refer to a memory address in terms of its distance from the instruction. At execution time, the Program Counter will contain a value 2 greater than the beginning of the instruction, that is, the address of the first byte of extension.

The 68000, 68HC001, 68008, 68302, 68010, 68331, and 68332 microprocessors may address **odd** memory locations only when the instruction is operating on a single byte. Neither the assembler nor the loader checks for this and in many cases (such as indexed address modes), neither the assembler nor the loader is capable of checking for this situation. The 68020/30/40 have no such restriction. However, all chips do require that every instruction begin at an even address, and the assembler enforces this. Data may begin at an even or odd address.

The remaining subsections briefly explain the particulars of both the 68000 model modes that apply to all 68000 family processors and the 68020 model modes that apply to the 68020 and later processors.

### **Register Direct Modes (I & II)**

Depending upon the mode, the Register Direct Modes act directly on the contents of either a data register or an address register.

All other modes specify an address in memory; the contents of this address are used as the instruction operand.

### **Address Register Indirect (III)**

The **Address Register Indirect Mode** provides the memory address in an address register.

### **Address Register Indirect with Postincrement (IV)**

The **Address Register Indirect with Postincrement Mode** provides the memory address in an Address Register and, after using the address, increments the register by one, two, or four, depending upon whether the scope of the operation is byte (.B), word (.W), or longword (.L).



### **Address Register Indirect with Predecrement (V)**

The **Address Register Indirect with Predecrement Mode** decrements an Address Register by one, two or four, depending upon whether the size of the operand is byte (.B), word (.W), or longword (.L), and then uses the resulting contents of the register as the memory address. None of the preceding modes require any extension bytes.

### **Address Register Indirect with (16-bit) Displacement (VI)**

In **Address Register Indirect with Displacement Mode**, the address is the sum of the contents of an address register and a sign-extended 16-bit displacement; it requires 2 bytes of extension.

### **Address Reg. Indirect with 8-Bit Displacement and Index (VI, 68000 model)**

In **Address Register Indirect with Displacement and Index Mode** the address is the sum of the contents of an Address Register, the contents of an Index Register (which may be an Address or a Data Register) and a sign-extended 8-bit displacement. It requires 2 bytes of extension. The Index Register involved may use either all 32 bits or 16 bits sign-extended.

### **Address Reg. Indirect with 8-Bit Displacement and Index (VIIa, 68332/020 model)**

In addition to the capabilities of the 68000 model, the 68332 model and the 68020 model allow the Index Register contents to be multiplied by a scale factor of 1, 2, 4, or 8 before being added to the Address Register contents. The scale factor is coded into bits 9 and 10 of the specially formatted extension word. In the 68000 model mode (VII), the scale factor is always 1.

### **Address Reg. Ind. with Base Displ. and Index (VIIb, 68332/020 model)**

The **Address Register Indirect with Base Displacement and Index Mode** calculates the memory address as the sum of the contents of an Address Register, the contents of an Index Register (which may be an Address or a Data Register) and a sign-extended base displacement which may be either 16 or 32 bits. This mode requires at least 2 bytes of extension, plus 2 more for a 16-bit displacement or 4 more for a 32-bit displacement. The Index Register involved may use either all 32 bits or 16 bits sign-extended. The Index Register

contents may be multiplied by a scale factor of 1, 2, 4, or 8 before being added to the Address Register contents. Any or all of the Address Register, Index Register and displacement may be specified to be null, in which case they are taken to have a value of 0. A null displacement does not require any extension bytes.

#### **Memory Indirect Post-Indexed (VIlc, 68020 model)**

The **Memory Indirect Post-Indexed Mode** first calculates an intermediate address as the sum of the contents of an Address Register and a sign-extended base displacement which may be either 16 or 32 bits. The final memory address is then calculated as the sum of the contents of the intermediate address, the contents of an Index Register (which may be an Address or a Data Register), and an outer displacement which may be either 16 or 32 bits. This mode requires at least 2 bytes of extension, plus 2 more for each displacement which is 16 bits and 4 more for each displacement which is 32 bits. The Index Register involved may use either all 32 bits or 16 bits sign-extended. The Index Register contents may be multiplied by a scale factor of 1, 2, 4, or 8 before being added to the intermediate address contents and the outer displacement. Any or all of the Address Register, Index Register, base displacement and outer displacement may be specified to be null, in which case they are taken to have a value of 0. Null displacements do not require any extension bytes.

#### **Memory Indirect Pre-Indexed (VIld, 68020 model)**

The **Memory Indirect Pre-Indexed Mode** first calculates an intermediate address as the sum of the contents of an Address Register, an Index Register (which may be an Address or a Data Register), and a sign-extended base displacement which may be either 16 or 32 bits. The final memory address is then calculated as the sum of the contents of the intermediate address and an outer displacement which may be either 16 or 32 bits. This mode requires at least 2 bytes of extension, plus 2 more for each displacement which is 16 bits and 4 more for each displacement which is 32 bits. The Index Register involved may use either all 32 bits or 16 bits sign-extended. The Index Register contents may be multiplied by a scale factor of 1, 2, 4, or 8 before being added to the Address Register contents and the base displacement. Any or all of the Address Register, Index Register, base displacement and outer displacement may be specified to be null, in which case they are taken to have a value of 0. Null displacements do not require any extension bytes.



### **Absolute Short (VIII)**

The Absolute Modes provide an actual memory address right in the instruction. For **Absolute Short Mode** this address is 16 bits sign-extended (2 bytes of extension). Because 16-bit addresses are sign-extended, the areas of memory addressable by Absolute Short Mode are from 0 to \$7FFF plus an area in high memory, the address range of which is dependent on the target chip (from \$FF8000 to \$FFFFFF for the 68000 and 68010, from \$F8000 to \$FFFF for the 68008, and from \$FFFF8000 to \$FFFFFFFF for the 68020/30/40).

Regardless of the target chip, the assembler recognizes only the absolute addresses from \$FFFF8000 to \$FFFFFFFF as being in the high short-addressable area of memory. (If it is necessary to use Absolute Short Mode on the actual area of high memory that is short-addressable on the target chip, any absolute code should be placed in a separate module and referenced as XREF.S from other modules, which technique causes the use of Absolute Short address mode in most cases. Alternatively such code could be made relocatable and placed in a SECTION.S, then located correctly at link time; in this case the high-short-addressable code need not be in a separate module.)

### **Absolute Long (IX)**

The Absolute Modes provide an actual memory address right in the instruction. **Absolute Long Mode** contains a full 32-bit address in the instruction and can thus address any memory location on any chip (4 bytes of extension).

### **Program Counter with Displacement (X)**

The **Program Counter Indirect with Displacement Mode** calculates the memory address by adding the value of the Program Counter to a sign-extended 16-bit displacement; it requires 2 bytes of extension.

### **Program Counter with 8-Bit Displacement and Index (XI, 68000 model)**

The **Program Counter Indirect with 8-bit Displacement and Index Mode** calculates the memory address by adding the value of the Program Counter, the contents of an Index Register (which may be Address or Data, and may use



the entire 32 bits or the low order 16 bits, sign-extended), and a sign-extended 8-bit displacement; it requires 2 bytes of extension.

**Program Counter with 8-Bit Displacement and Index (XIa, 68332/020 model)**

The 68332 model and 68020 model allow the Index Register contents to be multiplied by a scale factor of 1, 2, 4, or 8 before being added to the other components. The scale factor is coded into bits 9 and 10 of the specially formatted extension word. In the 68000 model mode (**XI**), the scale factor is always 1.

**PC with Base Displacement and Index (XIb, 68332/020 model)**

The **Program Counter Indirect with Base Displacement and Index Mode** calculates the memory address by adding the value of the Program Counter, the contents of an Index Register (which may be Address or Data, and may use the entire 32 bits or the low order 16 bits, sign-extended), and a sign-extended displacement, which may be either 16 or 32 bits. This mode requires at least 2 bytes of extension, plus 2 more for a 16-bit displacement or 4 more for a 32-bit displacement. The Index Register may be multiplied by a scale factor of 1, 2, 4, or 8 before being added to the other components. Any or all of the Address Register, Index Register, and displacement may be specified to be null, in which case they are taken to have a value of 0. A null displacement does not require any extension bytes.

**PC Memory Indirect Post-Indexed (XIc, 68020 model)**

The **Program Counter Memory Indirect Post-Indexed Mode** first calculates an Intermediate address as the sum of the contents of the Program Counter and a sign-extended base displacement which may be either 16 or 32 bits. The final memory address is then calculated as the sum of the contents of the Intermediate address, the contents of an Index Register (which may be an Address or a Data Register), and a sign-extended outer displacement which may be either 16 or 32 bits. This mode requires at least 2 bytes of extension, plus 2 more for each displacement which is 16 bits and 4 more for each displacement which is 32 bits. The Index Register involved may use either all 32 bits or 16 bits sign-extended. The Index Register contents may be multiplied by a scale factor of 1, 2, 4, or 8 before being added to the Intermediate address contents and the outer displacement. Any or all of the Program Counter, Index Register, base displacement and outer displacement



## Instructions and Address Modes

### Address Modes

may be specified to be null, in which case they are taken to have a value of 0. Null displacements do not require any extension bytes.

#### PC Memory Indirect Pre-Indexed (XIId, 68020 model)

The **Program Counter Memory Indirect Pre-Indexed Mode** first calculates an Intermediate address as the sum of the contents of the Program Counter, an Index Register (which may be an Address or a Data Register), and a sign-extended base displacement which may be either 16 or 32 bits. The final memory address is then calculated as the sum of the contents of the Intermediate address and a sign-extended outer displacement which may be either 16 or 32 bits. This mode requires at least 2 bytes of extension, plus 2 more for each displacement which is 16 bits and 4 more for each displacement which is 32 bits. The Index Register involved may use either all 32 bits or 16 bits sign-extended. The Index Register contents may be multiplied by a scale factor of 1, 2, 4, or 8 before being added to the Program Counter contents and the base displacement. Any or all of the Program Counter, Index Register, base displacement and outer displacement may be specified to be null, in which case they are taken to have a value of 0. Null displacements do not require any extension bytes.

#### Immediate (XII)

The final address mode provides data directly in the instruction (**Immediate Mode**). The number of bits used and the number of bytes of extension varies with the instruction and with the qualifier. Immediate data is always evaluated first as a 32-bit unsigned two's complement value. If the instruction requires fewer than 32 bits, the most significant bits are checked and discarded. If the bits discarded are all 0 or all 1, the instruction assembles normally, while if the bits discarded are mixed zeros and ones, a warning is printed. The immediate operands of ADDQ, SUBQ, TRAP, BKPT and all Shifts (which are smaller than a byte) may not be relocatable or external. All other immediate operands may be relocatable or external.

### 68881 Floating-Point Coprocessor and Address Modes

The 68881 floating-point coprocessor utilizes the 68020 addressing modes by requesting the 68020/30 to perform addressing mode calculations based on the 68881 instructions. **The 68881 knows nothing about addressing modes.** When instructed to do so by the 68881, the 68020/30 evaluates the instruction,

transfers the operands through the coprocessor interface, and performs the addressing mode calculations.

Any of the 68020 addressing modes may be used with floating point instructions, including **address/ data register direct, indexed indirect, auto increment, auto decrement, and immediate mode**. When a floating point instruction is encountered, the 68020 evaluates the instruction to its addressing modes. These include all 68020 addressing modes listed here, with the exception of a few restrictions for certain instructions. The exceptions are fully described in *Motorola Floating-Point Coprocessor User's Manual*.



### **68040 Floating-Point Unit and Address Modes**

The 68040 floating point unit uses the 68040 to perform address calculations. Thus any of the 68040 addressing modes may be used with floating point instructions.

## Assembler Syntax for Effective Address Fields

The assembler creates just one address mode for certain ways of specifying operands, while others may result in one of several modes. The following paragraphs describe how the Assembler makes such decisions. See table 2-6 for a definition of the terms which are used to describe operand syntax.

### Rules of Assembler Syntax

Motorola's 68020-oriented syntax is fully supported. This syntax uses square brackets "[,]" to designate the components of the intermediate address in the 68020 address modes, and parentheses to group the other components of an effective address. The following facts apply to address mode syntax:

- The syntaxes "< exp> (anything)" (old 68000) and "(< exp> ,anything)" (68020) are completely equivalent.
- The order of items separated by commas within square brackets or parentheses ("grouping characters") is not significant, unless there are two A-registers, neither having an appended size code nor scale factor, present within the same grouping characters. In this case (which is syntactically ambiguous) the leftmost register is taken as the Address Register and the rightmost as the Index Register.
- A 68000 model mode will be chosen if this is a possible interpretation of the operand, as these modes are more efficient. However, any of the following is sufficient to force a 68020 model address mode (perhaps with some null fields):
  - Using a Z-register (ZPC, ZAn or ZDn).
  - Using square brackets.
  - Specifying an explicit .L size code on a displacement. (Note that a .W qualifier does not force a 68020 model mode.) For example:  

```
( ( LABEL ) . L , A1 )
```
  - Specifying a scale factor other than 1 on an index register.
  - Specifying a displacement too large to fit in the 68000 model mode. Forward references are assumed to require 32 bits, while

## Instructions and Address Modes Assembler Syntax for Effective Address Fields

externals and relocatables are assumed to require 16 bits (but if the absolute part of an expression such as "reloc+ abs" is too large to fit in 16 bits, a 32-bit field will be used perforce). These defaults may be overridden by explicit .W and .L codes, and if a forward reference is later found to fit in 16 bits after all, a 68000 model mode may be selected on pass 2. (There will then be some extra NOPs trailing the instruction, however.) The OPT flags BRW and FRS do not apply to forward references which appear in conjunction with a register.

Note that coding, for example, "<exp> ,An)" rather than "<exp> (An)" is not sufficient to force the use of a 68020 model mode. Nor is specifying a scale factor of 1 explicitly. Errors will occur when assembler syntax forces 68020 model address modes and the target microprocessor (specified with the CHIP or OPT P= directives) is *not one that supports 68020 model addressing modes*.

- Assembler syntaxes which generate "Address Register Indirect with Displacement" or "Memory Indirect" modes (for example, "<exp> ,An)" or "([<exp> ,An],Rn)") allow <exp> to be an absolute or relocatable expression. If <exp> is an absolute expression, the assembler will use it as the displacement. If <exp> is a relocatable expression, the syntax says, "access the location of the relocatable expression using register 'An' indirect," and the linker/loader will calculate the final displacement. (See the "A2-A5 Relative Addressing" section for more information.)
- Absolute expressions in operands which generate Program Counter relative address modes (for example, "<abs exp> ,PC)") can have two different meanings depending on the ABSPCADD assembler flag.

By default, ABSPCADD is on, and the absolute expression is considered to be the address from which the current PC is subtracted to form the displacement.

When the ABSPCADD flag is off (OPT NOABSPCADD or OPT -ABSPCADD), the absolute expression is considered to be the displacement.

While you can use the OPT NOABSPCADD assembler option to code actual displacements in Program Counter relative instructions, there is also a way to specify actual displacements when the ABSPCADD flag is on. For example, if you would like to specify a displacement of + 8 from the current location counter, you could use the syntax "(+ 8,PC)" (which

Instructions and Address Modes  
**Assembler Syntax for Effective Address Fields**

is equivalent to OPT NOABSPCADD and the syntax "(6,PC)". The PC is 2 greater than the "\*" location counter symbol.)

In the tables that follow, the 68020 notation is used, but the facts listed above should be kept in mind. For example, the discussion of the operand "< abs exp> ,An,Rn{.W|.L})" includes the forms "< abs exp> (An,Rn{.W|.L})" and "< abs exp> ,Rn{.W|.L},An)".

### Operand Syntax and Addressing Modes

The following tables list what addressing modes the assembler will choose for the various operand syntaxes.

**Table 4-7. Definition of Syntax Terms**

SYNTAX TERM	DEFINITION
<b>An</b>	Represents an address register.
<b>Dn</b>	Represents a data register.
<b>Rn</b>	Represents either an address or data register, or a suppressed register (ZAn or ZDn). as68k does not recognize the mnemonic Rn.
<b>&lt; abs exp&gt;</b>	Represents an absolute expression, including an external reference with no section specified.
<b>&lt; rel exp&gt;</b>	Represents a relocatable expression, including an external reference with a section specified.
<b>&lt; exp&gt;</b>	Represents either an absolute or relocatable expression.
<b>{ }</b>	Represent a field that may or may not be present. (Note that the braces are required syntax in the 68020 BFxxx instructions, however.)

**Table 4-8. Operand Syntax & Addressing Modes**

<p><b>Dn</b> <b>An</b></p>	<p>The operands Dn and An always result in the <b>Data Register Direct (I)</b> and the <b>Address Register Direct (II)</b> modes, respectively.</p>
<p><b>(An)</b> <b>(An)+</b> <b>-(An)</b></p>	<p>The operands "(An)", "(An)+ " and "-(An)" always result in the <b>Address Register Indirect (III)</b>, <b>Address Register Indirect with Postincrement (IV)</b>, and <b>Address Register Indirect with Predecrement (V)</b> modes, respectively.</p>
<p><b># &lt; exp&gt;</b></p>	<p>This operand results in the <b>Immediate (XII)</b> mode. An absolute expression must be within a certain size range that is dependent on the instruction and qualifier code. 8-16- and 32-bit immediate data can be a relocatable expression.</p>
<p><b>(&lt; exp&gt; ,An)</b></p>	<p>This operand is resolved as <b>Address Register Indirect with Displacement (VI)</b>, provided the expression fits in 16 bits (sign-extended). The assembler assumes an external expression will fit into 16 bits.</p> <p>If the expression does not fit in 16 bits, the 68020 model mode <b>Address Register Indirect with Base Displacement and Index (VIIb)</b> is used. The specified A-register is used as the Address Register and the Index Register is taken to be null.</p> <p>As a special case, "(0,An)" generates the more efficient <b>Address Register Indirect (III)</b> despite the explicit zero displacement. A programmer who wishes to generate an explicit zero displacement will have to use an external symbol.</p>
<p><b>(Dn)</b> <b>(Rn.W)</b> <b>(Rn.L)</b> <b>(&lt; exp&gt; ,Dn)</b> <b>(,Rn{ .W  .L})</b></p>	<p>These operands generate the 68020 mode <b>Address Register Indirect with Base Displacement and Index (VIIb)</b>. The specified register is used as the Index register.</p>



**Table 4-8. Operand Syntax & Addressing Modes (Cont'd)**

<p>(&lt; abs exp&gt; ,An,Rn,{ .W  .L}{ *scl})          (An,Rn{ .W  .L}{ *scl})</p>	<p>If the target microprocessor is <i>not</i> the 68020/30/40 or 68331/332, the address mode generated is <b>Address Register Indirect with 8-Bit Displacement and Index (VII)</b>. The &lt; abs exp&gt; must resolve to an 8-bit, sign extended value. Otherwise, an error will occur. If the target microprocessor is the 68020/30/40 or 68331/332, the following cases determine the address mode generated:</p> <ol style="list-style-type: none"> <li>1. If &lt; abs exp&gt; is backward defined, its value fits in 8 bits, and the scale factor is 1, the <b>Address Register Indirect with 8-Bit Displacement and Index (VII)</b> 68000 model mode is generated. If the scale factor is greater than 1 (2, 4, or 8), then the 68020 model mode <b>VIIa</b> is generated.</li> <li>2. If &lt; abs exp&gt; is backward defined and its value is greater than 8 bits, the <b>Address Register Indirect with Base Displacement and Index (VIIf)</b> mode is generated.</li> <li>3. If &lt; abs exp&gt; is forward defined and its value fits in 8 bits and the scale factor is 1, the <b>Address Register Indirect with 8-Bit Displacement and Index (VII)</b> 68000 model mode is generated. If the scale factor is greater than 1 (2, 4, or 8), then the 68020 model mode <b>VIIa</b> is generated.</li> <li>4. If &lt; abs exp&gt; is forward defined and its value is greater than 8 bits, an error occurs because the assembler assumes that any forward defined absolutes will fit into 8 bits.</li> </ol> <p>If &lt; abs exp&gt; is absent, a displacement of 0 is used. Reading left-to-right, the first A-register found that does not have size code or scale factor is the Address register. The other register is the Index register.</p>
--	--



**Table 4-8. Operand Syntax & Addressing Modes (Cont'd)**

<p>( &lt; rel exp &gt; ,An,Rn{.W .L}{*scl})</p>	<p>If the target microprocessor is <i>not</i> the 68020/30/40 or 68331/332, this syntax always results in an error because the assembler did not allocate enough memory on the first pass.</p> <p>If the target microprocessor is the 68020/30/40 or 68331/332, this syntax results in the <b>Address Register Indirect with Base Displacement and Index (VIIb)</b>. If &lt; rel exp &gt; is forward defined, an error occurs because the assembler did not allocate enough memory on the first pass.</p> <p>Reading left-to-right, the first A-register found that does not have size code or scale factor is the Address register. The other register is the Index register.</p>
<p>([. . .],Rn . . .)          ([ &lt; exp &gt; ,An,Rn{.W .L}])</p>	<p>Any operand containing square brackets with a register specified outside the brackets (necessarily an Index Register), but not containing "PC" or "ZPC", generates the 68020 model <b>Memory Indirect Post-Indexed (VIIC)</b> mode. Any registers and displacements not specified are taken to be null. Any relocatable displacements are assumed to be 16 bits unless specified to be 32 bits by enclosing the expression in parentheses and attaching .L, i.e., (&lt; exp &gt; ).L.</p>
<p>([. . .,Rn], . . .)          ([ &lt; exp &gt; ,An,Rn{.W .L}])</p>	<p>Any operand which contains square brackets, with no register specified outside the brackets, and no "PC" or "ZPC" inside the brackets, generates the 68020 model <b>Memory Indirect Pre-Indexed (VIID)</b> mode. Any registers and displacements not specified are taken to be null. Any relocatable displacements are assumed to be 16 bits unless specified to be 32 bits by enclosing the expression in parentheses and attaching .L, i.e., (&lt; exp &gt; ).L.</p>
<p>(&lt; exp &gt; ,Dn,Rn{.W .L})          (Dn,Rn{.W .L})</p>	<p>These operands are invalid. One of the two registers must be an A-register or PC.</p>



Table 4-8. Operand Syntax & Addressing Modes (Cont'd)

<p>(&lt; exp&gt; ,PC)</p>	<p>This operand always results in <b>Program Counter Indirect with Displacement (X)</b> mode.</p> <p>If &lt; exp&gt; is an absolute expression, it is by default taken to be an address. The flag NOABSPCADD may be used to cause the absolute expression to be used as the displacement.</p> <p>If &lt; exp&gt; is an address, the displacement is calculated to be the value of &lt; exp&gt; minus the current value of the program counter. Sometimes, the assembler can calculate the displacement; in most cases, the calculation is postponed until link time when the actual location of both the instruction and the operand are known.</p>
---------------------------	---

**Table 4-8. Operand Syntax & Addressing Modes (Cont'd)**

<p>(&lt; exp&gt; ,PC,Rn{.W .L}{*scl})  (PC,Rn{.W .L}{*scl})</p>	<p>This operand results in modes <b>XI</b>, <b>XIa</b>, or <b>XIb</b> according to the following rules.</p> <ol style="list-style-type: none"> <li>1. If &lt; exp&gt; is relocatable. If &lt; exp&gt; is defined in the same section and the same source file as the instruction, the assembler can calculate the relative distance between &lt; exp&gt; and the instruction. Otherwise, the assembler cannot calculate the relative displacement and this calculation must be performed at link time. <ol style="list-style-type: none"> <li>a. If the assembler can calculate the displacement and this displacement will fit into 8 bits sign-extended, then mode <b>XI</b> is chosen.</li> <li>b. If the assembler can calculate the displacement, this displacement will fit into 8 bits sign-extended, and a scale factor greater than 1 is specified, then mode <b>XIa</b> is chosen.</li> <li>c. If the assembler cannot calculate the displacement or the displacement will not fit into 8 bits sign-extended, mode <b>XIb</b> is chosen.</li> </ol> </li> <li>2. If &lt; exp&gt; is absolute. <ol style="list-style-type: none"> <li>a. If the ABSPCADD flag is in effect and the instruction is also in an absolute section. In this case, the assembler can calculate the distance between &lt; exp&gt; and the instruction. <ul style="list-style-type: none"> <li>- If the displacement will fit into 8 bits sign-extended, mode <b>XI</b> will be chosen. A scale factor greater than 1 will cause mode <b>XIa</b>.</li> <li>- If &lt; exp&gt; is backward defined and the displacement is larger than 8 bits, mode <b>XIb</b> is chosen.</li> <li>- If &lt; exp&gt; is forward defined and the displacement is larger than 8 bits, an error will occur because the assembler did not allocate enough space on pass 1.</li> </ul> </li> </ol> </li> </ol>
---	--



**Table 4-8. Operand Syntax & Addressing Modes (Cont'd)**

	<p>b. If the ABSPCADD flag is in effect and the instruction is in a relocatable section.</p> <ul style="list-style-type: none"> <li>- If &lt; exp&gt; is backward defined, mode <b>XIb</b> is chosen.</li> <li>- If &lt; exp&gt; is forward defined, an error will occur because the assembler did not allocate enough space in pass 1.</li> </ul> <p>c. If the NOABSPCADD flag is in effect. If &lt; exp&gt; will fit into 8 bits sign-extended, mode <b>XI</b> is chosen. A scale factor greater than one will cause mode <b>XIa</b>. If &lt; exp&gt; will not fit into 8 bits, then mode <b>XIb</b> is chosen.</p>
--	---

<p>((&lt; exp&gt; ).W,PC,Rn{.W .L} { *scl})          ((&lt; exp&gt; ).L,PC,Rn{.W .L} { *scl})</p>	<p>A size qualifier on &lt; exp&gt; , e.g. (&lt; exp&gt; ).W or (&lt; exp&gt; ).L causes mode <b>XIb</b> to be chosen.</p>
<p>([. . .,PC],Rn,. . .)          ([&lt; exp&gt; ,PC],Rn{.W .L})</p>	<p>Any operand containing square brackets, with PC or ZPC inside, and a register specified outside the brackets (necessarily an Index Register), generates the 68020 model <b>Program Counter Memory Indirect Post-Indexed (XIc)</b> mode. When ZPC is used, the specified &lt; exp&gt; for the base displacement is always taken to be the displacement itself (in other words, the PC contents are not subtracted from it). At run-time, the PC is not used to create the effective address.</p>
<p>([. . .,PC,Rn],. . .)          ([&lt; exp&gt; ,PC,Rn{.W .L}])</p>	<p>Any operand which contains square brackets, with PC or ZPC inside, and no register specified outside, generates the 68020 model <b>Program Counter Memory Indirect Pre-Indexed (XIId)</b> mode. When ZPC is used, the specified &lt; exp&gt; for the base displacement is always taken to be the displacement itself (in other words, the PC contents are not subtracted from it). At run-time, the PC is not used to create the effective address.</p>

**< exp>**

The operand < exp> results in one of three modes: **Absolute Short (VIII)**, **Absolute Long (IX)**, or **Program Counter Indirect with Displacement (X)**. In most cases, good results will be obtained by allowing the assembler to use its default action.

---

**Note**

The PCR assembler flag (see the OPT assembler directive) controls the selection of addressing modes from a relocatable section to the same relocatable section in the same module.

You should note the following facts carefully before using the "< exp> " addressing modes table:

- The table does not apply to the Bcc or DBcc instructions, which use Program Counter plus Displacement mode.
- The final choice between address modes VIII and IX may be specified by the .S or .L qualifier on the JMP and JSR instructions. These qualifiers will not cause an absolute mode to be used instead of mode X, nor will they cause a reference to a location that is known to be in short-addressable memory to use absolute long mode.

The operand forms "< exp> .W" and "< exp> .L" are subject to the same rules as < exp> with the following clarifications:

- If an absolute (as opposed to a PC-relative) mode is chosen, "< exp> .W" forces the **Absolute Short (VIII)** mode and "< exp> .L" forces the **Absolute Long (IX)** mode.
- On forward references, "< exp> .W" forces 16 bits of extension to be allocated while "< exp> .L" forces 32 bits of extension to be allocated.

**Table 4-9. Choosing Address Modes for < exp >**

Instruction Section Type	Expression Type		
<b>ABS</b>	<b>&lt; abs exp &gt;</b>	<b>&lt; rel exp &gt;</b>	<b>unknown (forward ref)</b>
	<p>If OPT P is set and the displacement is within 16-bit range, then mode <b>X</b>.</p> <p>Else, if operand is in short addressable memory, then mode <b>VII</b>.</p> <p>Else mode <b>IX</b>.</p>	<p>If section of operand is short, then <b>VIII</b>, else <b>IX</b>.</p>	<p>If OPT F is set, then 2 bytes allocated, else 4 bytes allocated.</p>
<b>REL</b>	<p>If operand is in short-addressable memory, then <b>VIII</b>, else <b>IX</b>.</p>	<p>If OPT NOPCR is set, then if section of operand is short, then <b>VIII</b>, else <b>IX</b>.</p> <p>Else, if operand and instruction are in same section and displacement within 16-bits, then <b>X</b>.</p> <p>Else, if section of operand is short, then <b>VIII</b>, else <b>IX</b>.</p>	<p>If OPT F is set, then 2 bytes allocated, else 4 bytes allocated.</p>
<b>ABS</b>	External Reference in Specified Section		External Reference in Unspecified Section
	<p>If section of operand is short, then <b>VII</b>, else <b>IX</b>.</p>		<p>If operand was defined in XREF.S or if OPT F set, then <b>VII</b>, else <b>IX</b>.</p>
<b>REL</b>	<p>If OPT R set, then <b>X</b>.</p> <p>Else, if section of operand is short, then <b>VIII</b>, else <b>IX</b>.</p>		<p>If operand was defined in XREF.S or if OPT F set, then <b>VIII</b>, else <b>IX</b>.</p>

## How Code is Generated for Forward Defined Symbols

The assembler operates in two passes.

In Pass 1, when evaluating an operand, one or more labels may be forward defined. The assembler will not know whether these labels are absolute or relocatable symbols until later in the assembly. The assembler makes assumptions about forward defined labels; it selects a tentative address mode and allocates space for the instruction based on these assumptions.

In Pass 2, the assembler knows everything about the forward defined labels and will do one of three things:

- 1 It will generate the same addressing mode as it selected in Pass 1, and the space allocated for the instruction is exact.
- 2 It will see that a shorter, more efficient address mode could be used. It will generate the shorter address mode and fill the remaining allocated space with NOP instructions. The combination of the shorter address mode and the NOP instructions generally executes faster than the longer, less efficient address mode. For example, consider the instructions which follow.

```
Line Address
1 00000000 3038 1000 4E71      MOVE    F1,D0 ;Label F1 is
forward defined.
2
3 00001000      F1 EQU    1000H ;F1 may be
accessed using
4                                     ;absolute short
```

In Pass 1, the assembler assumes that the MOVE instruction will require the **Absolute Long (IX)** address mode. In Pass 2, the assembler sees that F1 may be accessed using the **Absolute Short (VIII)** address mode which requires only one word of operand extension. The assembler generates the MOVE instruction using the absolute short mode and fills the remaining word of the allocated instruction space with a NOP (4E71H) instruction.

The assembler flag `opnop` can be used to remove the NOP instructions that were used as filler. `opnop` causes the assembler to make additional passes through the code. This slows the assembly process but results in somewhat more compact code.

- 3 It will see that it did not allocate enough space in Pass 1 to generate the required instruction. An error will occur. For example, consider the instructions which follow.

## Instructions and Address Modes

### User Control of Address Modes

```
Line Address
1 00000000 3028 0000      MOVE  F1(A0),D0;Label F1 is forward defined.
  ** ERROR:(601) Value was truncated to fit in its field.
2
3 00020000      F1 EQU  20000H ;F1 is too large to be a
  ;16-bit displacement.
5                      END
```

In Pass 1, the assembler assumes that it will use the **Address Register Indirect with Displacement (VI)** mode which requires one word of operand extension. In Pass 2, the assembler determines that one word of extension is inadequate and an error occurs.

---

## User Control of Address Modes

The default choice for address mode is Absolute Long (in all cases except those where it is known that a more compact mode will work). Since this mode generates the longest machine codes (requiring 4 bytes of extension), you may want to choose a more compact and faster mode in some cases.

The choice of mode may be controlled in several ways:

- 1 Relocatable sections or external references may be specified as short (see the "Relocation" chapter for further information), meaning that any references to those sections and external references will use Absolute Short mode in preference to Absolute Long (but not in preference to other modes). Short sections and external references are always placed in the short-addressable areas of memory by the loader.
- 2 The option flag PCR may be set using the OPT directive. PCR (the default) causes references from a relocatable section to the same relocatable section in the same file to generate Program Counter with Displacement (X) mode if the displacement will fit into a signed 16-bit field. NOPCR causes such references to use absolute short or absolute long mode.
- 3 The option flag P may be set via the OPT directive, causing all references to a known absolute location from an absolute location to use Program Counter Indirect with Displacement mode, provided the displacement is within 16-bit range.



- 4 The option flag R may be set via the OPT directive, which causes all references from a relocatable location to a relocatable location (including external locations known to be in a relocatable section because the section name was specified with the XREF directive) to use Program Counter Indirect with Displacement mode. Most such references must be resolved by the loader. This option may cause assembler or linker errors if the referenced locations are not within a 16-bit displacement from the current PC.
- 5 The option flag F may be set via the OPT directive, causing all forward references except those in relative branch instructions (Bcc) to allocate only 2 bytes for the extension, rather than the default of 4 bytes. This option may result in errors at link-time, since it is possible that a location can only be addressed by Absolute Long mode, in which case there will not be room for the address and an error will result. With the default setting, however, even if 4 bytes are allocated, a 2-byte address mode may be selected finally (in accordance with the preceding table), in which case the final 2 bytes will be filled with a NOP.
- 6 The option flag B may be set via the OPT directive, which applies only to the relative branch instructions (Bcc) and causes forward references in one of these instructions to use the shorter form of the instruction, with 8-bit displacement. Here again it is possible that there may not be room for the actual displacement and errors may occur.
- 7 Individual Bcc, JMP and JSR instructions may use the .S or .L qualifiers on the opcode in order to force use of the short or long form of the instruction. In the Bcc instructions, use of these qualifiers forces the appropriate form. In the JMP and JSR instructions, use of these qualifiers does not force an absolute address mode to be chosen in those cases where a PC with displacement is known to work. However, if an absolute mode is used, the qualifier will force the choice of short or long, unless the reference is known to exist in short-addressable memory.

A Bcc.S instruction may not reference the next statement since this would result in an 8-bit displacement of 0, causing the hardware to take the following word as the 16-bit displacement, rather than as an instruction. Also, a Bcc.S may not reference an external reference or any location outside the instruction section (since the loader cannot resolve 8-bit displacements.)

## **A2-A5 Relative Addressing**

A2-A5 relative addressing refers to the method of accessing memory locations relative to an address in an address register. A2-A5 relative addressing is associated with the "address register indirect with displacement" addressing modes and the INDEX linker/loader command.

### **Address Register Indirect with Displacement Modes**

The "address register indirect with displacement" addressing modes are generated by operand syntaxes such as "< exp> (An)" or "(< exp> ,An,Rn)", etc. The displacements are calculated, if possible, by the assembler when "< exp> " is an absolute expression or by the linker/loader when "< exp> " is a relocatable expression.

### **Absolute Expressions vs. Relocatable Expressions**

When assembly language operands combine absolute expressions with address register indirection, the absolute expression is actually the displacement to be included with the instruction code.

When assembly language operands combine relocatable expressions with address register indirection (for example, < rel exp> (An) or (< rel exp> ,An)), the syntax says, "Access the location of the relocatable expression indirectly, using the address register." In other words, the relocatable expression is the effective address. When relocatable expressions are combined with address register indirection, the linker/loader will calculate the displacements with the following equation:

$$\text{<ea>} = \text{An} + \text{disp}$$

$$\text{disp} = \text{<ea>} - \text{An}$$

$$\text{disp} = \text{<relocatable expression>} - \text{An}$$

The linker/loader knows the value of the relocatable expression; however, it does not know what will be in "An" when the instruction executes.

To solve the linker's problem of not knowing the run-time contents of an address register (and allow you to use relocatable expressions in conjunction

with the powerful "address register indirect with displacement" modes), the linker/loader INDEX command was created to allow you to specify the run-time value of "An".

### **The INDEX Linker/Loader Command**

The INDEX loader command allows you to equate the run-time value of an address register (A2, A3, A4, or A5) with the load address of a relocatable section and an offset. The INDEX command will also create a public symbol in the form "?An" (where n = 2, 3, 4, or 5). The public symbol created can be declared as an external symbol in the assembly language source file (with the XREF directive) and used to initialize the appropriate address register.

When the INDEX command is not used, the linker will still calculate displacements for operands which combine relocatable expressions and address register indirection; however, the linker/loader will assume the run-time value of "An" to be zero.

### **Advantages of A2-A5 Relative Addressing**

A2-A5 relative addressing is useful when:

- Accessing statically allocated data areas. Accessing statically allocated data areas with A2-A5 relative addressing is as efficient as using the absolute short addressing mode with the additional benefit of being able to locate the data area (up to 64K bytes long) anywhere in memory.
- Accessing dynamically allocated data areas which are independent of the code that accesses them.

### **Accessing Statically Allocated Areas**

The 68000/20 model address register indirect with displacement addressing modes (for example, those modes generated for syntaxes such as "<exp> (An)" or "<exp> ,An,Rn", etc.) are often the fastest and most efficient ways to access code or data locations; this is especially true when accessing code or data in high memory where the alternative would be to use absolute long addressing (see figure 2-1). Notice that the address register indirect mode is coded in two fewer bytes than the absolute long mode.



## Instructions and Address Modes A2-A5 Relative Addressing

HPB3641-19300 A.02.00 27Apr93 Copr. HP 1988 Page 1 Wed Apr 28 15:21:21  
1993

Command line: as68k -L modes.s  
Line Address

```
1
2 00000000          WORD1 DS.W      SECT DATA
3 00000002          DS.B 0FFFEH    1
4
5                                     ; Address Mode Generated:
6 00000000 3039 0000 0000 R      MOVE WORD1,D0 ; Absolute Long.
7 00000006 302A 0000   R      MOVE WORD1(A2),D0 ; Address Reg. Indirect
8                                     ; with Displacement.
9
```

HPB3641-19300 A.02.00 27Apr93 Copr. HP 1988 Page 2 Wed Apr 28 15:21:21  
1993

### Symbol Table

Label	Value
WORD1	DATA:00000000

### Figure 4-1. Absolute and Indirect Address Modes

The address register indirect mode is useful because you can access locations anywhere in memory with the same number of bytes of code generated. Also, with a signed 16-bit displacement, you can access up to 64K bytes (+/- 32K) relative to the contents of the address register.

### **Accessing Dynamically Allocated Areas**

Dynamic memory allocation routines are typically passed the size of some element (for which memory is to be allocated) and return the address of the data area which has been allocated (in other words, a pointer to the allocated block of memory). At link-time, the linker/loader does not know what the address of the dynamically allocated area will be, but it does know the kind of element that memory is to be allocated for. With this knowledge, and with the help of the INDEX command, displacements can be calculated for A2-A5 relative addressing instructions. At run-time, the address of the dynamically allocated area is placed in the appropriate address register, and the dynamically allocated area can be accessed via A2-A5 relative addressing.



### **Example**

The following is a simple example of A2-A5 relative addressing and how to use the INDEX command. A listing of the assembly language source file is shown in figure 2-2. The linker/loader listing in figure 2-3 shows the INDEX command used with an offset. The linker/loader listing in figure 2-4 shows the INDEX command used without an offset. Comments are included in the assembly source file and in the linker command files to explain the instructions and commands in detail.

## Instructions and Address Modes

### A2-A5 Relative Addressing

HPB3641-19300 A.02.00 27Apr93 Copr. HP 1988 Page 1 Wed Apr 28 15:20:07

1993

Command line: as68k -L example.s

```

Line Address
1
2          XREF      ?A2 ; This symbol defined by the
3          ; linker/loader INDEX command.
4          XDEF      VAR ; (To get the effective address
5          ; on the linker/loader listing.)
6
7          SECT      DATA
8 00000000          DS.B 6000H
9 00006000          VAR DS.B 9FFFH ; Effective address of VAR =
10          ; load address of DATA =
11          ; section 6000H.
12
13          SECT      PROG
14 00000000 247C 0000 0000 E MOVE.L #?A2,A2 ; Initialize A2 with the
15          ; run-time value specified
16          ; in the INDEX command.
17
18 00000006 426A 6000      R CLR      VAR(A2) ;Address Register Indirect
19          ; with Displacement Mode
20          ; is generated. When this
21          ; module is linked, the
22          ; linker will calculate the
23          ; 16-bit displacement by
24          ; subtracting the run-time
25          ; value of A2 (as specified
26          ; by the INDEX command)
27          ; from the effective address
28          ; of VAR.
29          END

```

HPB3641-19300 A.02.00 27Apr93 Copr. HP 1988 Page 2 Wed Apr 28 15:20:07

1993

#### Symbol Table

Label	Value
?A2	External
VAR	DATA:00006000

**Figure 4-2. A2-A5 Relative Addressing Example**

## Instructions and Address Modes A2-A5 Relative Addressing

```
HPB3641-19300 A.02.00 27Apr93 Copr. HP 1988 Wed Apr 28 15:21:01 1993
Page 1
Command line: ld68k -c lnk_cmd.k -L
NAME lnk_cmd
LIST C ; Include a cross-reference listing on the output.
INDEX ?A2,DATA,8000H ; The run-time value of A2 equals the
; load address of the DATA section plus
; an offset of 8000H (this allows 16-bit
; signed displacements to access +/- 32K
; bytes relative to A2).
SECT DATA=0FF0000H ; Run-time of A2 is
; 0FF0000H + 8000H = 0FF8000H
; The displacement calculated for the "CLR VAR(A2)" instruction is
; the effective address of VAR (0FF0000 + 6000H) minus the run-time
; value of (0FF8000H):
;
; Displacement = 0FF6000H - 0FF8000H = -2000H = 0E000H.
;
; At run-time, the "MOVE.L #?A2,A2" instruction initializes A2
; with 0FF8000H. The "CLR VAR(A2)" instruction clears the location
; indexed by A2 plus the displacement, which equals:
;
; 0FF8000H + 0E000H = 0FF8000H + (-2000H) = 0FF6000H.
SECT PROG=1000H
LOAD example.o
END
HPB3641-19300 A.02.00 27Apr93 Copr. HP 1988 Wed Apr 28 15:21:01 1993
Page 2
OUTPUT MODULE NAME: lnk_cmd
OUTPUT MODULE FORMAT: IEEE
```

**Figure 4-3. Using the INDEX Command with Offset**

## Instructions and Address Modes

### A2-A5 Relative Addressing

#### SECTION SUMMARY

-----

SECTION	ATTRIBUTE	START	END	LENGTH	ALIGN
PROG	NORMAL CODE	00001000	00001009	0000000A	2 (WORD)
DATA	NORMAL DATA	00FF0000	00FFFFFFE	0000FFFF	2 (WORD)

#### MODULE SUMMARY

-----

MODULE	SECTION:START	SECTION:END	FILE
example	DATA:00FF0000 PROG:00001000	DATA:00FFFFFFE PROG:00001009	/users/merff/asm68k/example.o

#### CROSS REFERENCE TABLE

-----

SYMBOL	SECTION	ADDRESS	MODULE
?A2		00FF8000	-\$ example
VAR	DATA	00FF6000	-example

START ADDRESS: 00000000

Link Completed

**Figure 4-3. Using INDEX with Offset (Cont'd)**



## Instructions and Address Modes A2-A5 Relative Addressing

```
HPB3641-19300 A.02.00 27Apr93  Copr. HP 1988  Wed Apr 28 15:21:13 1993
Page 1
Command line: ld68k -c lnk_cmd2.k -L
NAME lnk_cmd
LIST C          ; Include a cross-reference listing on the output.
INDEX ?A2,DATA,0      ; The run-time value of A2 equals the
                    ; load address of the DATA section.
SECT DATA=0FF0000H   ; Run-time of A2 is 0FF0000H.
; The displacement calculated for the "CLR VAR(A2)" instruction is
; the effective address of VAR (0FF0000 + 6000H) minus the run-time
; value of (0FF0000H):
;
;           Displacement = 0FF6000H - 0FF0000H = 6000H.
;
; At run-time, the "MOVE.L #?A2,A2" instruction initializes A2
; with 0FF0000H. The "CLR VAR(A2)" instruction clears the location
; indexed by A2 plus the displacement, which equals:
;
;           0FF0000H + 6000H = 0FF6000H.
SECT PROG=1000H
LOAD example.o
END
HPB3641-19300 A.02.00 27Apr93  Copr. HP 1988  Wed Apr 28 15:21:13 1993
Page 2
```

```
OUTPUT MODULE NAME:   lnk_cmd
OUTPUT MODULE FORMAT: IEEB
```

**Figure 4-4. Using INDEX without Offset**

## Instructions and Address Modes A2-A5 Relative Addressing

### SECTION SUMMARY

-----

SECTION	ATTRIBUTE	START	END	LENGTH	ALIGN
PROG	NORMAL CODE	00001000	00001009	0000000A	2 (WORD)
DATA	NORMAL DATA	00FF0000	00FFFFFFE	0000FFFF	2 (WORD)

### MODULE SUMMARY

-----

MODULE	SECTION:START	SECTION:END	FILE
example	DATA:00FF0000 PROG:00001000	DATA:00FFFFFFE PROG:00001009	/users/merff/asm68k/example.o

### CROSS REFERENCE TABLE

-----

SYMBOL	SECTION	ADDRESS	MODULE
?A2		00FF0000	-\$ example
VAR	DATA	00FF6000	-example

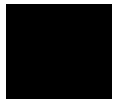
START ADDRESS: 00000000

Link Completed

**Figure 4-4. Using INDEX without Offset (Cont'd)**

---

# 5



---

## Relocation

This chapter explains relocatable programming and section attributes.

## Relocation

The object module produced by the assembler is in a relocatable format, which allows you to write programs whose final addresses will be adjusted by the linking loader. The relocatable format also allows individual program modules to be changed without reassembling the complete program. Separate object modules can be linked together into a final program.

Relocatable programming provides the following advantages:

- Actual memory addresses are of no concern until the final load time.
- Large programs may be easily separated into smaller pieces, developed separately, and linked together.
- If one piece contains an error, only that one need be modified and reassembled.
- Once developed, a library of routines may be used by many users.
- The linker will adjust addresses to meet program requirements.

## Program Sections

To take advantage of relocatability, you should understand the concept of program sections and how separate object modules are linked together. A program section is that part of a program which contains its own location counter and is a logically distinct section. At load time, the addresses for each section may be specified separately.

Section names may be any symbol or a two-digit decimal number. Section names may duplicate labels or register names without conflict. Section names may appear in COMMON, SECT (or SECTION) and XREF directives as well as in the .STARTOF. and .SIZEOF. operators.

as68k provides for up to 256 program sections. One section is predefined, noncommon section 0. Each section has five attributes: the common/noncommon attribute, the short/long attribute, the section contents attribute, the alignment attribute, and the HP Section type attribute.

### Common vs. Noncommon Attributes

A section becomes Common when its name appears in a COMMON directive, and becomes Noncommon when its name appears in a SECT or SECTION directive. It is a fatal error for the same section name to appear in both directives. The loader loads all common sections with the same name (from different modules) into the same place in memory, while noncommon sections with the same name (from different modules) are concatenated. Otherwise, Common and Noncommon sections are treated alike.

We suggest that you avoid putting instructions or code-generating directives (DC, DCB) in Common sections. If a user initializes the same Common section in two different modules, both sets of code will be loaded into the same memory locations by the Linker, and a warning is generated. This can obviously cause problems. On the other hand, initializing a Common section in only one module can be useful.

In a given assembly a section name may appear in an XREF directive before appearing in either a SECT (or SECTION) or COMMON directive. When this occurs, the assembler accepts the name as a valid new section name and assigns the Long or Short attribute to it as declared in the XREF directive, but does not yet assign the common or noncommon attribute to it.

## Relocation Program Sections

The common or noncommon attribute may be set by the subsequent occurrence of a SECT or COMMON directive that uses the same section name. However, if the current assembly does not assign the common/noncommon attribute, the linking loader may do so. In the latter instance, the section name must appear in a SECT or COMMON directive in another assembly; one whose object module is included in the load.

### Short vs. Long Attributes

A section becomes short when its name appears in a COMMON.S, SECT.S, SECTION.S, or XREF.S directive. It becomes long when its name appears in any of these directives without the .S extension. If a section is short in one place and long in another place, a warning is produced and the section is designated as short thereafter. The loader will load all short sections into the areas of memory addressable with 16-bit absolute addresses. These areas are from 0 to \$7FFF and from \$FF8000 to \$FFFFFF for the 68000 and 68010, \$F8000 to \$FFFFFF for the 68008, and \$FFFF8000 to \$FFFFFFFF for the 68020/30/40 and CPU32. (The linker CHIP command can specify a bus width parameter that could alter the location of the high short page.) In certain situations, the assembler will choose a more compact address mode when a reference is made to a short section (see the "Instructions and Address Modes" chapter for details). Otherwise short and long sections are treated alike.

### Section Alignment Attribute

The section alignment attribute may be either 1, 2 or 4. The section alignment attribute affects the beginning address of each file's contribution to a section.

A section alignment attribute of 4 combined with the ALIGN 4 directive can ensure that data items are located at longword boundaries. This may speed execution on some target systems where the memory bus is 32 bits wide.

The default section alignment attribute is 2 unless the CHIP directive specifies 68020/30/40, in which case the default attribute is 4. The alignment attribute is specified in the SECTION assembler directive as shown in the following example.

```
SECTION    A, 4
```

If the alignment attribute is specified differently in several files, the alignment attribute is affected in the following way:

- If there is no ALIGN linker command, the first module loaded in the section is always aligned modulo 4 if any of the modules for that section specify quad alignment. All other modules are aligned as specified by those modules.
- If there is an ALIGN linker command, all relocatable subsections of that section are aligned modulo the largest of the alignments.

## Section Contents Attributes

There are four types of relocatable sections:

- Program code (C).
- Data (D).
- Mixed Code and data (M).
- ROMable data (R).

The SECTION assembler directive allows you to explicitly specify a section's contents by adding a "C", "D", "M", or "R" qualifier to the SECTION directive. (See the SECTION description in the "Assembler Directives" chapter for details.)

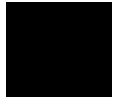
The section contents attribute is used by certain HP debuggers to gain efficiency and to prevent certain debugging commands from operating in particular areas of target memory.

The section contents attribute may be specified explicitly in the SECTION directive. For example:

```
SECTION          A,,C          ; Specifies a CODE section
```

If the section contents attribute is not specified explicitly, the assembler assigns the section type according to the following rules.

- 1 If, after the SECTION directive, the assembler encounters only instructions, the assembler will set the section contents attribute to program code (C).
- 2 If, after the SECTION directive, the assembler encounters only data definition directives (DC, for example), the assembler will set the section contents attribute to data (D).



## Relocation Program Sections

- 3 If, after the SECTION directive, the assembler encounters both instructions and data, the assembler will set the section contents attribute to mixed (M).

### HP Section Type Attribute

The HP 64000 symbolic files, `asmb_sym` and `link_sym`, supply program symbol information to HP 64000 emulators and analysis tools.

The HP Section type may be specified explicitly in the SECTION directive. For example,

```
SECTION A, ,C,P ;SECTION A MAPS TO HP PROG
```

The fourth operand of the SECTION directive may be P for PROG, D for DATA, C for COMN, or A for ABS.

If the HP type is not specified explicitly, the assembler uses the following rules.

- Program code (C) sections map to the HP 64000 section PROG.
- Data (D) sections map to the HP 64000 section DATA.
- ROMable data (R) sections map to the HP 64000 section COMN.
- "Extra" code, data, and ROMable data sections map to the HP 64000 section ABS (see below).
- Absolute (ORGed, in other words) sections map to the HP 64000 section ABS.

The HP 64000 assembler symbol and linker symbol file formats have the following characteristics.

- The file formats allow a maximum of three relocatable sections per assembly source file. For each assembly, at most one section may be mapped to PROG, one section may be mapped to DATA, and one section may be mapped to COMN.
- The file formats allow an unlimited number of absolute sections per assembly source file.

If the assembler, through any combination of SECTION directives, attempts to map more than one section onto PROG, DATA, or COMN using the rules



above, then this mapping conflicts with the HP 64000 file formats. The assembler and linker resort to the following stratagems.

- The second and subsequent sections that map to either PROG, DATA, or COMN are called "extra" CODE, DATA, & ROM sections.
- The symbols from "extra" sections are omitted from the HP 64000 assembler symbol file. This means that local (as opposed to global) symbols from extra sections will NOT be available at analysis time. When this happens, the assembler issues the following warning:

```
WARNING: (604) Manximum number of typed sections exceeded in HP mode.
```

- The code from "extra" section is correct and is treated normally.
- The linker, when producing a link\_sym file, maps the symbols from "extra" sections onto HP 64000 ABS sections. The symbol values are correct. They simply show up as ABS on HP emulators and analysis tools.

Because the HP 64870 assembler allows many relocatable sections, sometimes it is impossible to produce perfect HP 64000 assembler symbol and linker symbol files. In these situations, your code is always correct. At worst, you will not have access to some local symbols in some assembly files. You can overcome these limitations by moving "extra" sections to a different source file.

### Other Things to Know About Sections

Typically, a section will contain either instructions or data; this allows you to place the sections in a RAM/ROM environment. Common sections are generally used for program variables that reside in RAM. Common sections are analogous to named COMMON in FORTRAN. As with non-relocatable assemblers, users may also specify absolute addresses when assembling a program. In this case, the object modules, even if in relocatable format, will contain instructions or data that will reside in the specified memory locations.

### How the Assembler Assigns Section Attributes

Table 3-1 illustrates how a section is assigned the common/noncommon and short/long attributes. An example of how to use this table follows:

**Table 5-10. How Section Attributes are Assigned**

Previous Section Attribute	New statement in which section name appears:					
	XREF	XREF.S	SECT	SECT.S	COMMON	COMMON.S
Undefined	Xref-only LONG	Xref-only SHORT	Non-common LONG	Non-common SHORT	Common LONG	Common SHORT
Xref-only LONG	Xref-only LONG	Xref-only* SHORT	Non-common LONG	Non-common* SHORT	Common LONG	Common SHORT
Xref-only SHORT	Xref-only* SHORT	Xref-only SHORT	Non-common* SHORT	Non-common SHORT	Common* SHORT	Common SHORT
Common LONG	Common LONG	Common* SHORT	ERROR	ERROR	Common LONG	Common* SHORT
Common SHORT	Common* SHORT	Common SHORT	ERROR	ERROR	Common* SHORT	Common SHORT
Non-common LONG	Non-common LONG	Non-common* SHORT	Non-common LONG	Non-common* SHORT	ERROR	ERROR
Non-common SHORT	Non-common* SHORT	Non-common SHORT	Non-common* SHORT	Non-common SHORT	ERROR	ERROR

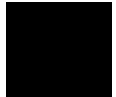
\* = Warning produced.

The first time a section name appears, it has no previous attributes; the first horizontal row of the table, marked undefined, is appropriate. If the name first appears in an XREF.S statement, it will afterwards be short, but neither common nor noncommon (XREF-only). If the name later appears for a second time in a SECT statement, it is then assigned the Noncommon attribute as well and a Warning is produced.

---

## Linking

The object modules produced by the assembler are combined or linked together by a linking loader. The loader converts all relocatable addresses into absolute addresses and resolves references from one module to another. Linkage between modules is provided by external definitions (XDEF), external references (XREF), as well as the Common Sections. External definitions are defined in other object modules via the linking loader. External references are symbols referenced in one module but defined in another module. The linking loader combines the external definitions from one program with the external references from other programs to obtain the final addresses. A program may contain both external references and definitions.



## Relocatable vs. Absolute Symbols

Each symbol in the assembler has associated with it a symbol type, which marks the symbol as absolute or relocatable. If relocatable, the type also indicates the section to which the symbol belongs. Symbols whose values are not dependent upon program origin are absolute, and those whose values change when the program origin is changed are called relocatable. Absolute and relocatable symbols may both appear in an absolute or relocatable program section.

*Absolute* symbols are defined as follows:

- A symbol in the label field of an instruction that is in an absolute section of code.
- A symbol is made equal to an absolute expression by the EQU or SET directive. This occurs even if the program is assembling a relocatable section.
- An external reference with no section attached is considered to be absolute for the purpose of determining address modes.
- The difference between two relocatable symbols if *both* symbols are defined in the same section in the same source file.

*Relocatable* symbols are defined as follows:

- A symbol in the label field of an instruction when the program is assembling a relocatable section.
- A symbol is made equal to a relocatable expression by the EQU or SET directives.
- An external reference is relocatable.
- A reference to the location counter (\*) while assembling a relocatable section is relocatable.

## Relocatable Expressions

The relocatability of an expression is determined by the relocation of the symbols that compose the expression. All numeric constants are considered absolute. Relocatable expressions may be combined to produce an absolute expression, a relocatable expression, or in certain instances, a complex relocatable expression. The following list shows those expressions whose result is relocatable: (ABS denotes an absolute symbol, constant, or expression and REL denotes a relocatable symbol or expression)

ABS+REL  
REL+ABS  
REL+REL  
REL-ABS<sup>1</sup>  
REL-REL<sup>1</sup>  
ABS\*REL  
REL\*ABS  
REL\*REL  
REL/ABS<sup>1</sup>  
REL/REL<sup>1</sup>

1. Absolute if both relocatable expressions are defined in the same section in the same source file. Otherwise, it is relocatable.

---

### Note

Complex relocatable expressions are not allowed in the ORG, OFFSET, COMLINE, END, FAIL, SPC, and LLEN directives.

Complex relocatable expressions result when two relocatable expressions are subtracted or added together. Only the plus "+" and minus "-" operators are allowed within these subexpressions. In certain instances, subexpressions may evaluate to an absolute value. This can occur in cases where a subexpression comprises the difference between two relocatable symbols.

After assembly has been completed, one of three types of expressions result:

- Absolute expression - The expression evaluates to an absolute value independent of any relocatable section addresses.

## Relocation

### Relocatable Expressions

- Simple relocatable expression - The expression evaluates to an absolute offset from a single relocatable section address.
- Complex relocatable expression - The expression evaluates to a constant absolute offset from either of the following:
  - A single, negated start address of a relocatable section.
  - References to the start address of two or more relocatable sections.

In addition, the following expressions are valid and produce an absolute expression. Both relocatable subexpressions must be relocatable in the same program section and must be defined in the current module (no externals).

```
REL=REL  
REL<>REL  
REL<=REL  
REL<REL  
REL>=REL  
REL>REL  
REL+REL  
REL-REL  
REL*REL  
REL/REL
```

---

## Label Alignment

Beware of labels on a line by themselves. They may not be aligned as you expect. For example,

```

                SECT    A
STRING DC.B    'odd'
START
                LEA    STACKTOP, SP

```

The label `START` will have an odd value. If the PC is loaded with an odd value, a run time error will occur.

There are two ways to avoid this problem:

- You may put the label on the same line as the instruction or directive. The label will have the same alignment as the instruction. For example,

```

                SECT    A
STRING DC.B    'odd'
START LEA    STACKTOP, SP

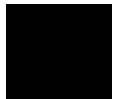
```

- You may also use an align directive after the byte constants. For example,

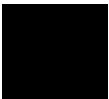
```

                SECT    A
STRING DC.B    'odd'
                ALIGN  2
START
                LEA    STACKTOP, SP `

```



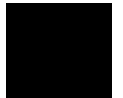
Relocation  
**Label Alignment**





---

# 6



---

## Assembler Directives

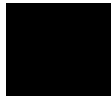
This chapter describes all directives (also called Psuedo-Ops) except those primarily associated with macro assembly and structured syntax.

## Assembler Directives

Assembler directives are written as ordinary statements in the assembler language, but rather than being translated into equivalent machine language, they are interpreted as commands to the assembler itself. Through use of these directives, the Assembler will reserve memory space, define bytes of data, assign values to symbols, control the output listing, etc. The following is a complete list of the directives that are described in this chapter.

ALIGN	Specify instruction alignment.
CHIP	Specify Target Microprocessor.
COMLINE	Define Storage.
COMMON	Specify Common Section.
DC	Define Constant Value.
DCB	Define Constant Block.
DS	Define Storage.
ELSEC	Conditional Assembly Converse.
END	End of Assembly.
ENDC	End Conditional Assembly.
ENDR	End Repeat.
EQU	Equate a Symbol to an Expression (permanent).
FAIL	Generate a Programmed Error.
FEQU	Equate a Symbol to a Floating Point Expression.
FILE	Include Source File (same as INCLUDE).
FOPT	Specify Floating-Point Options.
FORMAT	Format Listing (ignored).

IDNT	Specify Module Name.
IFC	Conditional Assembly String Equality Test.
IFDEF	Conditional Assembly Symbol Definition Test.
IFEQ	Conditional Assembly Equal to Zero Test.
IFGE	Conditional Assembly Nonnegative Test.
IFGT	Conditional Assembly Greater than Zero Test.
IFLE	Conditional Assembly Nonpositive Test.
IFLT	Conditional Assembly Less than Zero Test.
IFNC	Conditional Assembly String Inequality Test.
IFNDEF	Conditional Assembly Symbol Not Defined Test.
IFNE	Conditional Assembly Unequal to Zero Test.
INCLUDE	Include Source File.
INTFILE	Specify File for Intermediate Storage.
IRP	Specify Indefinite Repeat.
IRPC	Specify Indefinite Repeat Character.
LIST	List the Assembly.
LLEN	Set Length of Line in Assembler Listing.
MASK2	Assemble for R9M chip (ignored).
NAME	Specify Module Name.
NOFORMAT	Don't Format Listing (ignored).

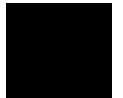


## Assembler Directives

NOLIST	Don't List the Assembly.
NOOBJ	Don't Create Object File.
NOPAGE	Suppress Paging on Listing.
OFFSET	Define Table of Offsets.
OPT	Set Options for Assembly.
ORG	Begin an Absolute Section.
PAGE	Advance Listing Form to Next Page.
PLEN	Specify Length of Listing Page.
REG	Define a Register List.
REPT	Specify Repeat.
RESTORE	Restore previously SAVEed assembly options.
SAVE	Save assembler options.
SECT	Specify Section.
SECTION	Specify Section.
SET	Equate a Symbol to an Expression (temporary).
SPC	Space lines on listing.
TTL	Set Program Heading.
XCOM	Specify Weak External Reference.
XDEF	Specify External Definition.
XREF	Specify External Reference.

## Notation

In the following descriptions, brackets ( { } ) are used to indicate optional parameters. If more than one item appears within a single pair of brackets, a choice is indicated.



## ALIGN

### Specify Byte Alignment

#### Syntax:

Label	Operation	Operand	Comment
	ALIGN	n	

#### Where:

n Equals either 1, 2 or 4.

#### Description:

This directive may be used to specify the byte boundary on which the address of the next instruction is to be aligned. The number may be either 1, 2 or 4.

The ALIGN directive is useful for adjusting the location counter to the nearest word or longword boundary.

Modulo 4 alignment can be used to optimize execution speeds, depending on the target system memory design.

However, in order for modulo 4 alignment to work in a relocatable section, you must first ensure the alignment of a section when it is located by the linker. This is done by specifying a section alignment attribute of 4 in the SECTION directive. See the following example.

#### Example:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	SECTION	A,4	;The beginning of every file's ;contribution to a section will be ;quad aligned.
	DC.B	'A number of characters'	
LABEL1	ALIGN	4	;Ensure next data item is ;quad aligned.
Q1	DC.L	0	;Q1 is on a mod 4 boundary.

#### See also

Other aspects of section and label alignment are discussed in chapter 5.

## CHIP

### Specify Target Microprocessor

**Syntax:**

Label	Operation	Operand	Comment
	CHIP	target	

**Where:**

target	Is one of the following processor designations: 68000, 68EC000, 68HC000, 68HC001, 68008, 68010, 68302, 68330, 68331, 68332, 68333, 68340, CPU32, 68020, 68EC020, 68030, 68EC030, 68040, or 68EC040.
--------	---

**Description:**

This directive specifies the microprocessor on which the resulting object code will be run. The microprocessor may be the 68000, 68EC000, 68HC000, 68HC001, 68008, 68010, 68302, 68330, 68331, 68332, 68333, 68340, CPU32, 68020, 68EC020, 68030, 68EC030, 68040, or 68EC040. The differences, from the assembler's point of view, are as follows:

- 1 The 68010 has the additional instructions MOVECaN INDIC, MOVES, RTD and MOVE from CCR. If one of these instructions is encountered when the CHIP is set to 68000 or 68008, code for the instruction is generated, but an error occurs.
- 2 The 68020 has the additional instructions BFCHG, BFCLR, BFEXTS, BFEXTU, BFFFO, BFINS, BFSET, BFTST, BKPT, CALLM, CAS, CAS2, CHK2, CMP2, DIVSL, DIVUL, PACK, RTM, TDIVS, TDIVU, TRAPcc, Tcc, TPcc, and UNPK. It has six new address modes as described in the INSTRUCTIONS AND ADDRESS MODES chapter. The Bcc, BSR, DIVS, DIVU, EXTB, LINK, MOVEC, MULS, MULU and TST instructions accept additional qualifiers and/or operands. Using any of these constructs when the CHIP is not set to 68020 causes an error. Note that using new 68020 syntax is not sufficient to cause an error, provided the generated code is 68000-compatible. Examples of this include an explicit \*1 scale factor on an index register, using the EXTB and EXTW synonyms for EXT, placing a displacement inside rather than

## Assembler Directives

### CHIP

outside the delimiting parentheses, and rearranging the order of registers inside parentheses.

- 3 The 68331 and 68332 have, in addition to 68010 capabilities, the additional instructions BGND, CHK2, CMP2, EXTB, LPSTOP, TBLs, TBLU, TBLSN, TBLUN, TRAPcc, Tcc, and, TPcc. It has new addressing modes as described in the Instructions and Addressing Modes chapter. The Bcc, BSR, DIVS, DIVU, LINK, MULS, MULU, and TST accept additional qualifiers and/or operands.

The 68331 and 68332 do not have a co-processor interface. Therefore, CHIP 68332 (68331) disables the 68881 FPU instructions.

- 4 The 68030 has the additional instructions PFLUSH, PFLUSHA, PLOADR, PLOADW, PMOVE, PMOVEFD, PTESTR, and PTESTW. It also has the additional registers CRP, SRP, TC, TT0, TT1, and MMUSER.
- 5 The 68040 has the same instructions as the 68020/30 with the addition of CINVL, CINVP, CINVA, CPUSHA, CPUSHL, CPUSHP, MOVE16, PFLUSHAN, and PFLUSHN.

If no CHIP or OPT P= (which has the same function) directive appears, the target is assumed to be the 68000.

Using new Motorola 68020, 68030, or 68040 syntax is not sufficient to produce a warning, provided the generated code is 68000-compatible. Examples of new syntax are explicit scale factor on an index register, using the EXTB and EXTW synonyms for EXT, placing a displacement inside rather than outside delimiting parentheses, and rearranging the order of registers inside parentheses.

Chip designations are now processed as strings, which means using absolute expressions with the CHIP directive is no longer valid.



---

## COMLINE

### Define Storage

#### Syntax:

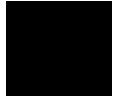
Label	Operation	Operand	Comment
{label}	COMLINE	n	

#### Where:

n                    The number of bytes of memory to be reserved.

#### Description:

This directive may be used in the source code to reserve a block of sequential locations (in bytes). The number of bytes is specified in the argument (e.g., COMLINE 8 reserves 8 bytes in memory). COMLINE is supplied for Motorola compatibility. as68k treats this directive identically to DS.B.



## COMMON

### Specify Common Section

**Syntax:** There are 3 distinct syntaxes:

Label	Operation	Operand	Comment
{label}	COMMON{.S}	sname[,n][,[contents][,HPtype]]	
	COMMON{.S}	snumber[,n][,[contents][,HPtype]]	
label	COMMON{.S}	snumber[,n][,[contents][,HPtype]]	

**Where:**

sname	The name of the COMMON section.
snumber	A one or two digit decimal number used to construct the COMMON section name.
n	Alignment for this module section. May be 1, 2 or 4.
contents	An indication of the contents of this module section. May be M (mixed code & data), C (code), D (data), or R (ROMable data).
HPtype	How to map this section onto HP 64000 symbol files. May be P (PROG), D (DATA), C (COMN), or A (ABS).

**Description:**

The COMMON directive specifies to the assembler that the following statements should be assembled in the relocatable mode using the named common section specified. This section remains in effect until an ORG, SECT, SECTION, OFFSET, or another COMMON directive is assembled that specifies a different section. Initially all section location counters are set to zero.

The user may alternate between various sections with multiple SECT and COMMON directives within one program. The assembler will maintain the current value of the location counter for each section.

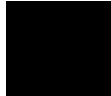
The common section name may be any symbol or a two-digit decimal number. The label field has different meanings in these two cases.

In all cases, the optional .S determines whether or not the section has the short attribute. In the first case, "sname" is the name of the specified common section, and "label", if present, will be assigned the address of the current location counter; in other words, it is a normal label. In the second case, "snumber" is a two-digit decimal number which is the name of the common section. In the third case, "snumber" is a one or two digit decimal number, and "label" is combined with "snumber" to produce the name of the common section.

Note that the same section name or number should not appear in both a COMMON and a SECT directive, except where a label is placed on a numbered section to create a named common area. Note also that relocatable section 0 is predefined to have the noncommon attribute and thus may not appear in an unlabeled COMMON directive.

**Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
LABEL1	COMMON	SECT1	;name is SECT1, LABEL1 is ;normal symbol
	COMMON	CODE	;name is CODE
	COMMON	1	;name is 1, common section
LABEL1	COMMON	1	;name is 1LABEL1, common section. ;No conflict with other LABEL1
	COMMON	C1,4,D,D	;name is C1 ;alignment mod 4 ;contents r/w data ;maps to HP DATA section



## DC

### Define Constant

#### Syntax:

Label	Operation	Operand	Comment
{label}	DC{.qualifier}	operand1{,operand2,...}	

#### Where:

label	An optional label that will be assigned the address of the first byte defined.
qualifier	May be <code>.B</code> for byte data, <code>.W</code> for word data, <code>.L</code> for longword data, <code>.S</code> for single-precision floating, <code>.D</code> for double-precision floating, <code>.P</code> for packed decimal floating, or <code>.X</code> for extended-precision floating. Default is <code>.W</code> .
operand	For qualifiers <code>.B</code> , <code>.W</code> and <code>.L</code> , a character string or an expression. All expressions are calculated as 32-bit values. For <code>.B</code> , this value must fit in 8 bits (either 0-filled or one-filled); for <code>.W</code> , it must fit in 16 bits. If this condition is violated, a warning is produced. For qualifiers <code>.S</code> , <code>.D</code> , <code>.P</code> , and <code>.X</code> , a floating-point number is required. A floating-point number which cannot be stored in the indicated number of bits (because its exponent is too large) is reported as an error. However, excessive bits of precision in a specified mantissa are truncated without a warning.

#### Description:

The DC directive is used to define up to 509 bytes of data. For operands other than character strings, the assembler will allocate one byte per operand for a DC.B, two bytes per operand for a DC.W or DC with no qualifier, four bytes per operand for a DC.L or DC.S, eight bytes per operand for a DC.D, and twelve bytes per operand for a DC.P or DC.X. All operands (except character strings) must evaluate to a value that fits in this number of bytes or an error is generated. Negative values are stored using their two's complement representation. Operands of a DC.W or DC.L may be relocatable; operands

of a DC.B may not be. Operands of a DC.S, DC.D, DC.X, or DC.P may only be floating-point numbers.

Character strings are stored one character per byte, starting at the lowest-addressed byte. Character strings in a DC.W or DC.L are padded out with zeroes in the least significant bytes of the last words, if necessary, to bring the total number of bytes allocated to a multiple of 2 or 4, respectively.

If an odd number of bytes is entered in a DC.B directive, the odd byte on the right will be skipped and the Location Counter aligned to an even value, unless the next statement is another DC.B, a DS.B or a DCB.B. The byte skipped over is not initialized in any way.

The .S and .D qualifiers permit definition of Single and Double precision floating-point numbers respectively. The generated bit patterns are IEEE standard and compatible with the Motorola MC68881 coprocessor, and also with the 68040 on-chip coprocessor. Single precision is 1 sign bit, 8 exponent bits (biased by 127), and 23 mantissa bits. Double precision is 1 sign bit, 11 exponent bits (biased by 1023), and 52 mantissa bits.

The .X qualifier permits the definition of an Extended precision floating-point number. The .P qualifier permits the definition of a Packed Decimal floating-point number.

Floating point numbers may be in either decimal or hexadecimal format. A decimal floating-point number must contain either a decimal point or an "E" indicating the beginning of the exponent field. For example: "3.14159", "-22E-100". The latter example means "-22 times (10 to the -100th power)". Underscores may occur before or after the "E" to increase readability. Underscores are ignored in determining the value of a constant.

A hexadecimal floating point number is denoted by a colon ":" followed by a series of hex digits: up to 8 digits for single-precision, or 16 digits for double-precision. The digits specified are placed in the field as they stand; the user is responsible for determining how a given floating-point number is encoded in hexadecimal digits. If fewer digits than the maximum permitted are specified, the ones that are present will be **left**-justified within the field. Thus the first digits specified always represent the sign and exponent bits.

The DC.S, DC.D, DC.X, and DC.P directives will accept only floating-point numbers as operands. DC with any other qualifier will not accept floating-point numbers as operands.

Assembler Directives  
DC

**Example (generated  
bytes shown):**

```
4142 4344 4566   DC.B 'ABCDEFghi'
6768 69
45               DC.B 'E'           ; starts at odd address
6500            DC   'e'
4500 0000       DC.L 'E'
3132 3334 3500   DC.L '12345'
0000
000A 0005 0007   DC.W 10,5,7
00FF           DC   $FF
3F80 0000       DC.S 1.0
3FF0 0000 0000   DC.D 1.0
0000
3F80 0000       DC.S :3F8
3FF0 0000 0000   DC.D :3FF
0000
3FF0 0000 0000   DC.P 1.0
0000 0000 0000
3FF0 0000 0000   DC.P :3FF
0000 0000 0000
3FFF 8000 0000   DC.X 1.0
0000 0000 0000
3FFF 8000 0000   DC.X :3FFF8
0000 0000 0000
```

---

## DCB

### Define Constant Block

#### Syntax:

Label	Operation	Operand	Comment
{label}	DCB{.qualifier}	length,value	

#### Where:

label	An optional label that will be assigned the address of the first byte allocated.
qualifier	Defines the units in which storage is measured. May be <code>.B</code> for bytes, <code>.W</code> for words, <code>.L</code> for longwords, <code>.S</code> for single-precision floating, <code>.D</code> for double-precision floating, <code>.P</code> for packed-decimal floating, or <code>.X</code> for extended-precision floating. Default is <code>.W</code> .
length	An absolute expression defining the number of units of storage to allocate. The expression may not contain forward, undefined or external references.
value	The value to which each unit is initialized. For qualifiers <code>.B</code> , <code>.W</code> and <code>.L</code> , this is an expression that may contain forward references, relocatables, externals or complex expressions. For qualifiers <code>.S</code> , <code>.D</code> , <code>.P</code> , and <code>.X</code> , this is a floating-point number as described under the DC directive.

#### Description:

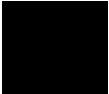
The DCB directive causes the assembler to allocate a block of bytes, words, longwords, single-precision floating numbers (32 bits) or double-precision floating numbers (64 bits) depending on the qualifier. Each unit allocated is set to the same given value. This directive causes the location counter to be aligned to a word boundary, unless the `.B` qualifier is specified.

Use of the DCB directive causes the assembler to generate a byte of code for each byte (not unit) allocated. This can lead to large object files.

Assembler Directives  
**DCB**

**Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	DCB.L	100,\$FFFFFFF	





## DS

### Define Storage

#### Syntax:

Label	Operation	Operand	Comment
{label}	DS{.qualifier}	size	

#### Where:

label	An optional label that will be assigned the address of the first byte allocated.
qualifier	Defines the units in which storage is allocated. May be <code>.B</code> for bytes, <code>.W</code> for words, <code>.L</code> for longwords, <code>.S</code> for single-precision floating, <code>.D</code> for double-precision floating, <code>.P</code> for packed-decimal floating, or <code>.X</code> for extended-precision floating. Default is <code>.W</code> .
size	A value that specifies the number of units to be allocated by this directive. Any symbols used in this expression must be previously defined. The final expression may not contain any relocatable terms.

#### Description:

This directive is used to reserve a block of sequential locations of memory. It causes the program counter to be advanced. The contents of the reserved bytes are unpredictable. Locations may be reserved in units of bytes, words, longwords, single-precision floating numbers (32 bits), double-precision floating numbers (64 bits), extended precision floating-point numbers (96 bits), or packed binary coded decimal floating-point numbers (96 bits).

The Define Storage (DS) directive causes the location counter to be aligned to a word boundary unless the `.B` qualifier is used. The form `DS 0` may be used to force alignment between two `DC.B`, `DS.B` or `DCB.B` statements, if necessary.

Assembler Directives  
**DS**

**Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
JAKE	DS	\$62	
MOE	DS.B	100	



## **ELSEC**

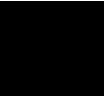
### **Conditional Assembly Converse**

The ELSEC directive is used in conjunction with one of the conditional assembly directives (IFNE, IFEQ, IFLT, IFLE, IFGE, IFGT, IFC, or IFNC) and is the converse of the conditional assembly directive. When the argument of the conditional assembly directive evaluates to false, all statements between the ELSEC directive and the next ENDC are assembled. When the argument of the conditional assembly directive evaluates to true, no statements between the ELSEC directive and the next ENDC are assembled.

The ELSEC directive is optional and can only appear once within a conditional block.

### **Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	IFNE	MAIN	
	-		
	ELSEC		
	-		
	ENDC		



## **END**

### **End of Assembly**

#### **Syntax:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	END	{expression}	

#### **Where:**

expression      An address that is placed in the end record of the load module and informs the loader where program execution is to begin. If this expression is not specified, the module is considered not to contain a starting address. If no module read by the loader contains a starting address, execution begins at absolute 0. If {expression} is not present but a comment field is present, the latter must be preceded by semicolon (;) or exclamation mark (!).

#### **Description:**

The END directive is used to inform the assembler that the last source statement has been read and to indicate a load module starting address. Any statements following the END directive will not be processed.

Specifying a load address in this directive also informs the loader that this is a main program. If multiple load modules are combined by the Linking Loader, only one module may specify a load address and hence be a main program.

#### **Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	END	MAIN	

## ENDC

### End Conditional Assembly Code

**Syntax:**

Label	Operation	Operand	Comment
	ENDC		

**Description:**

The ENDC directive is used to inform the Assembler where the source code subject to the conditional assembly statement ends. In the case of nested IFxx statements, an ENDC is paired with the most recent IFxx statement.

In the following code, if the expression SUM-4 is equal to zero, the instructions between the IFEQ and ELSEC directives will not be assembled and those between the ELSEC and ENDC will be assembled. If SUM-4 is non-zero, the opposite occurs. To inhibit listing the non-assembled instructions the OPT -I directive may be used.

**Example:**

Label	Operation	Operand	Comment
	MOVE	#22,D2	
	IFEQ	SUM-4	
	ORI	#200,D3	;assembled if
	ADD	D0,VALUE+3	;SUM-4 is zero
	ELSEC		
	ORI	#\$1F,D3	;assembled if
	ROL	#1,D0	;SUM-4 is non-zero
	ENDC		

## **ENDR**

### **End Repeat**

#### **Syntax:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	ENDR		

#### **Description:**

The ENDR directive is used to end a repeat statement as defined by the REPT, IRP, or IRPC directives. Note that an ENDR does not terminate a macro definition.

#### **Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	IRP	D1	
	ADD	D0 , VALUE+3	
	ENDR		

## EQU

### Equate a Symbol to an Expression

**Syntax:**

Label	Operation	Operand	Comment
label	EQU	expression	
label	EQU	keyword	
label	EQU	externsymbol[+ offset]	
label	EQU	externsymbol[-offset]	

**Where:**

label	A symbol defined by this statement.
expression	An expression whose value will be assigned to the given label for the duration of the current assembly. An attempt to re-equate the same label will result in an error. Any symbols used in the expression must be defined previously.
keyword	A keyword defined by the assembler or a symbol previously defined by this directive as a keyword. Keyword may also be a simple forward reference.
externsymbol	An externally defined symbol (XREF).
offset	A constant integer value.

**Description:**

The EQU directive causes the assembler to assign a particular value to a new label, which may be an absolute or a relocatable section value (see the "Relocatable Symbols" section in the "Relocation" chapter). It may also be a single external symbol. In the case of an external symbol, you may add or subtract a constant value from the label.

Simple forward references (a single symbol with no operators) are now accepted by EQU.

## Assembler Directives

### EQU

EQU may also be used to define new keywords to be used instead of the predefined assembler keywords, which allows the user to assign meaningful names to processor registers.

#### Example:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
SEVEN	EQU	D7	
INDEX	EQU	A5	

---

**Note** The following example illustrates the misuse of an EQU.

---

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
Reg	EQU	D0	
	MOVE	#0,0(A0,Reg.L)	; "Reg.L" causes error.
Reg.L	EQU	D0.L	
	MOVE	#0,0(A0,Reg)	; No error occurs.



## FAIL

### Generate Programmed Error

**Syntax:**

Label	Operation	Operand	Comment
	FAIL	{expression}	

**Where:**

expression	If present, should be absolute and contain no forward references. If absent, 0 is used. If the value of {expression} is less than 500, FAIL produces error number 591. If {expression} is greater or equal to 500, FAIL produces a warning number 591.
------------	--

**Description:**

The FAIL directive may be used to indicate an error or warning. The typical place for this directive is within convoluted nestings of macros and conditional assemblies, to mark a path of assembly that would never be taken if the code did what the user intended. If the value of {expression} is less than 500, FAIL produces error number 591. If {expression} is greater or equal to 500, FAIL produces a warning number 591. When a FAIL directive is assembled, the assembler marks it with a "Fail encountered" error or warning message and displays the 32-bit value of the directive's argument in the address field of the listing.

## FEQU

### Equate a Symbol to a Floating Expression

#### Syntax:

Label	Operation	Operand	Comment
label	FEQU{.qual}	fp-expression	

#### Where:

label	A symbol defined by this statement.
qual	May be .S for single-precision, .D for double-precision, .X for extended-precision, or .P for packed-decimal.
fp-expression	An floating-point expression whose value will be assigned to the given label for the duration of the current assembly. An attempt to re-equate the same label will result in an error. Any symbols used in the expression must be defined previously.

#### Description:

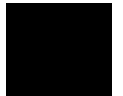
The FEQU directive allows the assembler to assign a floating-point expression to a symbol. as68k supports the IEEE standard floating-point number format with the exponent section being optional.

Floating point numbers may be in either decimal or hexadecimal format. A decimal floating-point number must contain either a decimal point or an "E" indicating the beginning of the exponent field. For example: "3.14159", "-22E-100". The latter example means "-22 times (10 to the -100th power)". Underscores may occur before or after the "E" to increase readability. Underscores are ignored in determining the value of a constant.

A hexadecimal floating point number is denoted by a colon ":" followed by a series of hex digits: up to 8 digits for single-precision, or 16 digits for double-precision. The digits specified are placed in the field as they stand; the user is responsible for determining how a given floating-point number is encoded in hexadecimal digits. If fewer digits than the maximum permitted are specified, the ones that are present will be **left**-justified within the field. Thus the first digits specified always represent the sign and exponent bits.

**Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
COUNT1	FEQU	123.45	
COUNT2	FEQU.X	:9AB	



## **FILE**

### **Specify Include File**

See the description for the INCLUDE directive later in this chapter.



---

## FOPT

### Specify Floating-Point Options

#### Syntax:

Label	Operation	Operand	Comment
	FOPT	ID= n	

#### Where:

n                    A number in the range 0 through 7 specifying the coprocessor ID field.

#### Description:

The FOPT directive specifies the coprocessor ID field (0 through 7) used in subsequent 68881 floating-point instructions. If no FOPT directive is specified, the default ID is 1 (the 68881 coprocessor).

#### Example:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	FOPT	ID=2	; Specify 68881 ID #2.
	FMOVE.D	#2.0,FP0	; Move to 68881 ID #2.
	FOPT	ID=1	; Specify 68881 ID #1.
	FMOVE.D	#2.0,FP0	; Move to 68881 ID #1.

## **FORMAT, NOFORMAT**

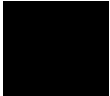
### **Format the Listing**

#### **Syntax:**

Label	Operation	Operand	Comment
<hr/>			
[NO]FORMAT			

#### **Description:**

These directives are recognized for Motorola compatibility but are ignored by the assembler. as68k does not require them but recognizes them for compatibility with the Motorola directives **FORMAT** and **NOFORMAT**. Motorola uses these directives to format or not to format the source listing.



---

## **IDNT**

### **Specify Module Name**

#### **Syntax:**

Label	Operation	Operand	Comment
name	IDNT		

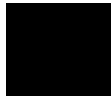
#### **Where:**

name            The name to be placed in the object module denoting the module name to the loader. This name must follow all the rules of a symbol. This name appears in the label field of the statement. The operand field of the statement is ignored.

#### **Description:**

The IDNT directive is used to assign a name to the object module produced by the assembler. It is identical in function to the NAME directive; however, IDNT allows only legal identifiers for the module name, while NAME allows an arbitrary sequence of characters. Only one IDNT directive should appear in a program.

If an IDNT or NAME directive is not specified by the user, the default name is the input file name (without path and extension).



## IFEQ, IFNE, IFGT, IFGE, IFLT, IFLE

### Conditional Statements Comparing to Zero

#### Syntax:

Label	Operation	Operand	Comment
	IFxx	expression	

#### Where:

expression      Evaluates to a value that determines whether or not the assembly between the IFxx and the following ELSEC or ENDC will take place. Any symbols used in this expression must be previously defined. The expression may not be relocatable.

#### Description:

The IFxx directive may be used to conditionally assemble source text between the IFxx directive and the ELSE or ENDC directive. When the expression in the operand field is in the indicated relationship to zero, the code will be assembled. IFxx statements may be nested up to 16 levels and appear at any place within the source text.

Note that these directives perform a signed comparison, treating their operands as two's complement 32-bit signed integers ranging from -\$80000000 to +\$7FFFFFFF. In contrast, the logical operators >, <= and so forth perform unsigned comparisons, treating their operands as 32-bit unsigned integers ranging from 0 to +\$FFFFFFFF. Therefore "IFGT X" is not equivalent to "IFNE X> 0". Logical operators return a value of \$FFFFFFFF for TRUE and zero for FALSE.

#### Example:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	IFGE	RAMBASE	



---

## IFC, IFNC

### Conditional Assembly String Equality Test

#### Syntax:

Label	Operation	Operand	Comment
	IFC	{string1},{string2}	
	IFNC	{string1},{string2}	

#### Where:

string1            Are defined below.  
string2

#### Description:

The IFC and IFNC directives provide a way to test whether two strings are or are not equal. Depending on the result of the comparison, following statements up to the next ELSEC or ENDC will or will not be assembled (like the IF statement). These directives take two string arguments, both optional, separated by a required comma. The strings are defined as follows (where the term "nonblank" excludes tab characters also):

- If the first nonblank character following the directive is a comma, the first string is null.
- If the first nonblank character following the directive is a single quote, the first string consists of all characters from this quote to the matching closing quote, including the delimiting quotes. As usual, two adjacent quotes represent a quote character within the string. In this case, the next nonblank after the closing quote must be a comma and blanks between the closing quote and the comma are not significant. Commas may appear between the quotes as part of the string.
- If the first nonblank character following the directive is neither a comma nor a single quote, the first string consists of all characters from this one to the last nonblank before the first comma on the line. The comma is not part of the string. An unbalanced quote may be part of a string in this format. Note that a string in this format cannot contain commas.
- The first string is always terminated by a comma, which is referred to below as the "delimiting comma".

## Assembler Directives IFC, IFNC

- If there are no nonblanks after the delimiting comma, the second string is null.
- If the first nonblank after the delimiting comma is a semicolon, the second string is null.
- If the first nonblank after the delimiting comma is a single quote, the second string goes from this quote to the terminating quote, as for the first string. Any characters after the terminating quote are ignored.
- If the first nonblank after the delimiting comma is not a single quote or a semicolon, the second string goes from the first nonblank following the delimiting comma to the last nonblank before the first semicolon following the delimiting comma; or, if there is no semicolon following the delimiting comma, to the last nonblank on the line. In this format, the first semicolon after the delimiting comma is considered a comment delimiter; it and all characters after it are ignored. Note that in this format, the second string may not contain semicolons.

### Examples:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	IFC	'STRING','STRING'	;equal--assembly continues
	IFNC	'string',' string'	;unequal (blank in 2nd string) ;assembly continues
	IFC	A'\1',A'\2'	;always unequal
	IFC	'\1','\2'	;parameters are expanded
	IFC	\1,\2	;parameters are expanded
	IFC	string , string	;equal (blanks not significant)

## **IFDEF, IFNDEF**

### **Conditional Assembly Symbol Definition Test**

**Syntax:**

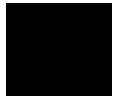
Label	Operation	Operand	Comment
	IFDEF, symbol		
	IFNDEF, symbol		

**Where:**

symbol                      Is a program symbol that may or may not have been defined or declared external. No forward references are allowed.

**Description:**

The IFDEF and IFNDEF directives provide a way to test if a symbol has been defined or declared external. Depending on the result of the test, following statements up to the next ELSEC or ENDC will or will not be assembled. These directives take a single symbol as an argument that cannot be a forward reference.



## **INCLUDE**

### **Include Source File**

#### **Syntax:**

Label	Operation	Operand	Comment
	INCLUDE	filename	
	FILE	filename	

#### **Where:**

filename      The name of the host computer file to be inserted in the Assembly Source File.

No lower to upper case conversion is performed on file names. If the file name has a suffix ("x.h", for example), the file name is passed without change to the operating system. If the file name has no suffix ("source", for example), then the suffix ".s" is appended to the filename before it is passed to the operating system.

#### **Description:**

The INCLUDE (FILE) Directive may be used to insert an external source file into the input source code stream at Assembly time. Include statements may not be nested and have some limitations when combined with macro calls. A macro call may contain an INCLUDE directive, but, if an INCLUDE file is invoked by a macro call, the INCLUDE file may not contain any additional macro calls.

The default search directory (when none is explicitly specified), is the current directory. Additional search paths may be specified on the command line. See the **as68k** syntax in the on-line manual pages.

#### **Example:**

Label	Operation	Operand	Comment
	INCLUDE	EXTERNAL.S	
	FILE	EXTERNAL.S	

---

## **[NO]INTFILE**

### **Sorts Information Using Intermediate File or Virtual Memory**

**Syntax:**

Label	Operation	Operand	Comment
	[NO]INTFILE		

**Description:**

The linker, like the assembler, is a two pass program. Intermediate information is stored, by default (for non-PC hosts), using virtual memory between pass 1 and 2. The INTFILE command lets you store this intermediate information in a temporary file. The NOINTFILE command lets you store this information using virtual memory. Use this command if ERROR 340 occurs.

With different systems, using a temporary file may be faster than using virtual memory. Also, depending on the configuration for running large jobs, the virtual allocation size can be limited if a virtual error is returned and error message (ERROR 340) is displayed.

Using the INTFILE command is the same as specifying the -b option on the command line.

**Example:**

```
INTFILE  
LOAD mod1.obj  
END
```

## IRP

### Specify Indefinite Repeat

#### Syntax:

Label	Operation	Operand	Comment
{label}	IRP	model parameter{,actual parameter, ...}	

#### Where:

label	An optional label assigned the address of the current program counter.
model parameter	The parameter which will be replaced by actual parameters.
actual parameter	The actual parameter whose number determines the number of repeats.

#### Description:

The IRP directive specification includes a "model" parameter followed by a list of actual parameters. The sequence of statements enclosed by the IRP and ENDR directives is repeated once for each actual parameter, substituting the actual parameter everywhere the model is found. Parameter substitution is identical to that which is performed in a macro.

The parameter list begins after the model parameter. A null parameter list causes the macro to be expanded one time with a null replacing the model parameter.

Like macro definitions, repeat directives cannot be nested. Only one macro definition may be used inside a repeat directive.

#### Example:

Label	Operation	Operand	Comment
	IRP	DUMMY, SUB1, SUB2, SUB3	
	JSR	DUMMY	;Three JSR instructions generated
	ENDR		

## IRPC

### Specify Indefinite Repeat Character

**Syntax:**

Label	Operation	Operand	Comment
{label}	IRPC	model parameter{,actual parameter }	

**Where:**

- label                    An optional label assigned the address of the current program counter.
  - model parameter      The parameter which will be replaced by actual parameters.
  - actual parameter     The actual parameter whose length determines the number of repeats.
- Each character in the parameter will be substituted for the model parameter during each repetition.

**Description:**

The IRPC directive specifies a model parameter and a single actual parameter. The sequence of statements is repeated once for each character of the actual parameter. The IRPC directive may be terminated with the ENDR directive.

The actual parameter list begins after the first parameter. A null actual parameter list causes the macro to be expanded one time with a null replacing the model parameter.

Like macro definitions, repeat directives cannot be nested. Only one macro definition may be used inside a repeat directive.

**Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	IRPC	DUMMY,1234	
	MOVE	#DUMMY,D0	;Four MOVE and JSR instructions
	JSR	SUB	;are generated.
	ENDR		

## **LIST**

### **Turn On Source Listing**

Label	Operation	Operand	Comment
	LIST		

**Description:**

This directive causes a listing of the assembly to be printed. This is the default. (The OPT S directive is another way to indicate a listing of the assembly is to be printed.)





---

## **LLEN**

### **Change Length of Output Listing Line**

**Syntax:**

Label	Operation	Operand	Comment
	LLEN	n	

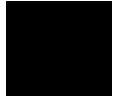
**Where:**

n                    An absolute expression whose value is between 37 and 1100 inclusive. Forward references are not allowed.

**Description:**

This directive changes the length of the line on the source listing. The user specifies the new length, which must be between 37 and 1100 inclusive. The value of 116 allows printing of the full 80 columns of the input source. The default value for the line length is 132.

This directive does not affect the header lines at the top of each page, which are printed in a format of fixed length.



## **MASK2**

### **Generate Code for R9M**

This directive is recognized for Motorola compatibility but is ignored.



---

## **NAME**

### **Specify Module Name**

#### **Syntax:**

Label	Operation	Operand	Comment
	NAME	modulename	

#### **Where:**

modulename      The name to be placed in the object module denoting the module name to the loader.

Modulename is specified as an arbitrary sequence of characters from the first non-white-space character following NAME through the end of the line.

---

#### **Note**

Because the modulename is everything up to the end of the line, a comment field used with the NAME directive will cause the comment to become part of the modulename.

---

#### **Description:**

The NAME directive is used to assign a name to the object module produced by the assembler. It is identical in function to the IDNT directive. However, the syntax of NAME allows the module name to be an arbitrary sequence of characters while IDNT allows only legal identifiers. Only one NAME or IDNT directive should appear in a program.

If a NAME or IDNT directive is not specified, the default module name is the input file name (without path and extension).

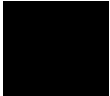
## **NOLIST**

### **Turn Off Source Listing**

Label	Operation	Operand	Comment
	NOLIST		

**Description:**

This directive suppresses printing of the assembler listing. The OPT -S directive may also be used to suppress the listing.



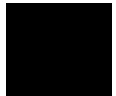
---

## **NOOBJ**

### **Suppress Creation of Output Object Module**

Label	Operation	Operand	Comment
	NOOBJ		

**Description:** This directive suppresses creation of the output object module. The OPT -O directive may also be used for this purpose.



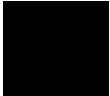
## **NOPAGE**

### **Suppress Paging on Listing**

Label	Operation	Operand	Comment
	NOPAGE		

**Description:**

This directive suppresses all page ejects and page headers on the output listing, including those explicitly specified by the PAGE directive. NOPAGE affects the entire listing, no matter where the directive appears in the program. Once paging has been disabled it cannot be re-enabled.



## OFFSET

### Define Table of Offsets

Label	Operation	Operand	Comment
{label}	OFFSET	n	

### Where:

- label            An optional label to identify the offset location.
- n                An absolute expression containing no forward references.

### Description:

This directive is used to define a table of absolute offsets. It is present for convenience and compatibility, but performs no function that cannot be handled with EQU's. The OFFSET directive is much like ORG in that it terminates the previous section and alters the Location Counter to an absolute value. However, an OFFSET "section" may not contain code and instructions, DC and DCB directives are illegal within an OFFSET section. OFFSET has one operand, which is an expression that must be absolute and must contain no forward or external references. This required operand is the new value for the Location Counter. The OFFSET "section" is terminated by an ORG, OFFSET, SECT, SECTION, COMMON or END directive.

The usual use for OFFSET is to define a storage template in mnemonic terms. For example, suppose we want to define symbols to represent the beginnings of the 80 column rows of an 80 column by 24 row character terminal screen. Suppose further that we define an area of memory called SCREEN and that we will address the rows as SCREEN+ ROW1, SCREEN+ ROW2, and so on.

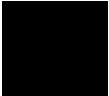
The following example uses EQU to achieve the purpose.

Label	Operation	Operand	Comment
SCREEN	DS.B	80*24	
ROW1	EQU	SCREEN+0	
ROW2	EQU	SCREEN+80	
ROW3	EQU	SCREEN+160	
.			
.			
ROW24	EQU	SCREEN+1840	

A clearer alternative for complex structures is the use of OFFSET.

Assembler Directives  
**OFFSET**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
SCREEN	DS.B	80*24	
	OFFSET	0	
ROW1:	OFFSET	80	
ROW2:	OFFSET	80	
.			
.			
ROW24:	OFFSET	80	
	END		





## OPT

### Set the Options Specified

**Syntax:**

Label	Operation	Operand	Comment
	OPT	[no  -]flag { ,[no  -]flag }...	

Precede the flag with "-" or NO to turn the flag off.

**Where:**

flag is one of the following:

**ABSPCADD** (Absolute w/PC = Address) specifies that an absolute expression appearing in conjunction with the mnemonic "PC" refers to an address rather than an absolute displacement. Thus 5(PC) would refer to absolute address 5, reached via PC relative mode, rather than 5+ current PC. This flag applies to the base displacement, not the outer displacement, in the 68020/30/40 expressions containing square brackets. This flag may be turned on and off at the user's discretion; the last setting applies. (Default= ABSPCADD)

**B** (Branch) specifies that forward references in relative branch instructions (Bxx) will use the short form of the instruction (8-bit displacement). This option affects only Bxx instructions (Default= BR W, or 16 bit displacements).

**BRL** (Branch) forces the long address mode to be used in relative branch instructions (Bcc, BRA, BSR) that have forward references. In 68020/30/40 mode, 32-bit displacements will be used unless OPT OLD has been specified. If OPT OLD has been specified with 68020/30/40 mode, 16 bit displacements will be used. In all other processor modes, 16 bit displacements are used. (Default= BR W, or 16 bit displacements).

Assembler Directives  
**OPT**

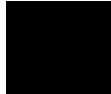
---

**Note** NOB, NOBRB, NOBRS, and NOBRL all cause BRW to be chosen.

---

BRW	(Branch) forces 16 bit displacements to always be used in relative branch instructions (Bcc, BRA, BRS) that have forward references (Default= BRW, or 16-bit displacements.)
C CEX	Specifies that all lines of data (after the first) generated by the DC directive will be listed. NOTE: this option does not affect the operation of the DC directive. CEX is a synonym for C (Default= C).
CASE	Specifies that symbols are case sensitive (Default= CASE).
CL I	Lists instructions not assembled due to conditional assembly statements (Default).
CRE X	Specifies that the cross reference information appears as an addition to the symbol table information (Default= NOCRE).
D	(Debug) specifies that the symbol table will be placed into the object module and may be used for debugging. This option must also be specified before any instruction that generates object code. If OPT CASE is set, symbols are placed in symbol table as defined. (Default= D).
E	(Error) specifies that Error messages and Warnings will be listed on the standard error output (Default= E).
F FRS	Causes the assembler to allocate 16 bits for extensions on instructions whose operands contain forward defined symbols. F is a synonym for FRS and -F or NOF is a synonym for FRL (Default= FRL).
FRL	Causes the assembler to allocate 32 bits for extensions on instructions whose operands contain forward defined symbols. During Pass 2, the assembler may decide to access the operand using absolute-short, absolute-long, or PC-plus-displacement modes (Default= FRL).

G	List the assembler generated symbols in the symbol or cross reference table. If D is also on, these symbols are placed in the object module as well (Default= NOG).
HLASYM	Affects the symbolic information in the IEEE relocatable file for compiler-generated modules. <b>HlasyM</b> causes assembly- level local symbols to be put into the output file. <b>NohlasyM</b> causes assembly-level local symbols from compiler-generated modules resulting in smaller output files. Compiler-generated symbols are not affected by this flag (default: <b>nohlasyM</b> ).
M MEX	(Macro expansion) specifies that macro expansions and structured syntax expressions will be listed in the program listing. MEX is a synonym for M (Default= M).
MC	(Macro calls) specifies that macro calls will be listed in the program listing (Default= MC).
MD	(Macro Definitions) specifies that macro definitions will be listed in the program listing (Default= MD).
NEST= n	Sets the maximum nesting level of macros to <i>n</i> . The default is the maximum level for nesting (Default= 100).
O	(Object) specifies that the object module will be produced (Default= O).
OLD	Specifies that the interpretation of the OPT BRL directive, and explicit .L qualifiers on Bcc instructions, will be 16-bit displacements (as appropriate for the 68010 and earlier processors), even though the processor mode is 68020 or greater. This is convenient for migrating 68000 programs onto the 68020/30/40 and CPU32 chips. (Default= NOOLD).
OP= n	Sets the maximum number of optimization loops that the assembler will do if OPT OPNOP is set. The assembler will discontinue looping either when there is a pass in which no



Assembler Directives  
**OPT**

optimization occurs or when this limit is reached  
(Default= OP= 3).

**OPNOP** Remove NOP instructions generated by the assembler. When the assembler encounters a forward reference during pass 1, it will allocate space for an instruction based on worst case assumptions. During pass 2, it will sometimes generate a shorter form of the instruction and fill the remaining space with NOPs. This flag removes those NOPs but at the cost of increased assembly time because it makes additional passes over the file (Default: **noopnop**).

**P** (Program counter relative) specifies that a program counter with displacement address mode will be used on references within the absolute section, provided that this address mode is legal for the instruction and that the displacement from the program counter fits within the 16-bit field provided. This option does not affect references either from or to a relocatable section. **PCO** is a synonym for **P** (Default= **NOP**).

**P= chip** (Processor type) identifies the target processor. This option is distinguished from **OPT P** by the equals sign, which must immediately follow the **P**. See the **CHIP** directive (which is equivalent to **OPT P=** ) for a list of valid processor types and for a discussion of the differences between the various target processors. The preceding **NO** or minus sign is not permitted on this option, because it makes no sense (Default= 68000).

**PCR** (PC Relative) specifies that a program counter plus displacement address mode will be used on references from a relocatable section to the same relocatable section. This applies to all instructions for which the program counter relative address mode is legal, provided that the displacement fits into the 16-bit field.

**PCS** (Relocatable) specifies that a program counter with displacement address mode will be used on references from a relocatable section to a relocatable section. This applies to all instructions for which program counter relative is a

legal address mode. The PCS flag applies to references to a different section within a file and to all external references that have any relocatable section name specified. If R is on and a reference to a relocatable section results in a displacement larger than 16 bits, it is considered an error. PCS is a synonym for R (Default= NOR).

- QUICK Quick allows the assembler to optimize certain mnemonics when possible. The mnemonic optimizations are MOVE to MOVEQ, ADD to ADDQ, and SUB to SUBQ. NOQUICK prevents these optimizations (Default= QUICK).
- REL32 This flag applies to 68020/30/40 address modes.
- Rel32** causes the assembler to use 32-bit base and outer displacements for forward, external, or relocatable operands.
- Norel32** causes 16-bit base and outer displacements. This flag applies to operands that do not have explicit word or longword size qualifiers (Default: **norel32**).
- S (Source) specifies the source text will be listed. The directives LIST and NOLIST are other ways to specify OPT S and OPT -S respectively (Default= S).
- T (Table) specifies the symbol table will be listed (Default= T).
- W Specifies that warnings are to be suppressed during the assembly (Default= NOW).

**Description:**

The OPT directive may be used to generate listings of the elements specified, to influence the assembler's choice of address modes in ambiguous situations, and to control the form of the object output.

The defaults in the assembler are:

- The source text, symbol table, macro definitions, macro calls, macro expansions, and conditional assembly statements not assembled are all listed.
- An object module in relocatable format is produced.

## Assembler Directives

### OPT

- The symbol table is placed into the object module.
- References to locations whose relative displacement cannot be determined at assembly time will use an absolute address mode, unless the user specifically requests otherwise.
- Forward and external references will leave room for an absolute long address.
- A relative branch to a forward reference will use the long (16-bit displacement) form of the instruction.
- The target chip is the 68000.
- The 68881 instructions are legal.
- Symbols are case sensitive.
- Error messages and warnings are listed to the standard error output.

To turn on an option, use the single or multiple letter code shown below. (Many options have more than one possible spelling.) To turn off an option, precede it by a minus sign or the characters "NO". Default settings for options are shown below.

Error messages are always listed, regardless of the elements specified. In particular, the E option may be used to generate a listing that consists only of error messages and is in a separate file.

### Example:

Label	Operation	Operand	Comment
	OPT	-x,D	do not list cross reference table but put symbol table in object module

## ORG

### Begin Absolute Section

#### Syntax:

Label	Operation	Operand	Comment
	ORG	{.qualifier} {expression} {,name}	

#### Where:

qualifier	Maybe S or L. ORG.S is interpreted as both ORG and OPT FRS. ORG.L is interpreted as both ORG and OPT FRL. ORG with no qualifier does not alter the F option.
expression	A value that will replace the contents of the Assembly Location Counter; bytes subsequently assembled will be assigned memory addresses beginning with this value. This expression may contain no forward, undefined, or relocatable symbols (including external references). The form " <i>* + or- displacement</i> " is legal. The value of <i>*</i> in this case is the ending value of the previous absolute section, or 0 for the first absolute section.
name	Specifies the name of the section.

If a comment field is present, it must be preceded by semicolon (;) or exclamation mark (!).

#### Description:

The ORG directive is used to begin an absolute section. The Location Counter is set to the value of the operand, if present. If there is no operand, the Location Counter is set to immediately follow the last preceding absolute section, if there was one. If the first ORG in a program has no operand, the Location Counter is set to 0. All subsequent bytes will be assigned sequential addresses beginning with the address in the Location Counter.

If the program does not have an ORG, SECT, SECTION or COMMON statement before the first code-generating statement, a SECTION 0 is assumed and assembly begins at location zero in the relocatable noncommon long section named 0.

Assembler Directives  
**ORG**

**Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	ORG	\$100	





---

## **PAGE**

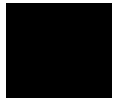
### **Advance Listing Form to Next Page**

**Syntax:**

Label	Operation	Operand	Comment
PAGE			

**Description:**

This directive instructs the assembler to skip to the top of the next page on the listing form, in order to make program listings easier to read. Some programmers prefer to start each subroutine on a new page. If the NOPAGE directive was specified, this directive is ignored.



## **PLEN**

### **Specify Length of Listing Page**

**Syntax:**

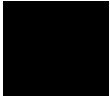
Label	Operation	Operand	Comment
	PLEN	n	

**Where:**

n                    An absolute expression whose value is greater than 12.

**Description:**

PLEN specifies the number of lines in an assembler listing page. The default value is 60. The value specified must be greater than 12.



## REG

### Define Register List

#### Syntax:

Label	Operation	Operand	Comment
label	REG	register-list	

#### Where:

label	A symbol whose value is to be defined.
register-list	List of registers in the format recognized by the MOVEM instruction, to wit: <ul style="list-style-type: none"> <li>- A single register.</li> <li>- A range of consecutive registers of the same type (A or D), denoted by the lowest and highest registers to be transferred separated by a hyphen (lower one must come first).</li> <li>- Any combination of the above separated by a slash.</li> </ul>

#### Description:

This directive assigns a symbolic name to a register list for future use by the MOVEM instruction. The symbol may be redefined as a different register list. Note that this redefinition is not compatible with Motorola.

#### Example:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
SAVE	REG MOVEM	A1-A5/D0/D2-D4/D7 (A6),SAVE	

## REPT

### Specify Repeat

#### Syntax:

Label	Operation	Operand	Comment
{label}	REPT	count	

#### Where:

label	An optional label assigned the address of the current program counter.
count	An expression indicating the number of times to repeat the code. This expression may not be relocatable or contain symbols not previously defined.

#### Description:

This directive allows a sequence of directives to be repeated a specified number of times. The statements to be repeated are those between the REPT and the following ENDR directive. The statements are expanded from the point at which the REPT directive is encountered.

#### Example:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	REPT	3	;Repeat next 2 lines 3 times.
	DC.B	'A'	
	DC.B	'B'	
	ENDR		

---

## RESTORE

### Restore Assembler Options

#### Syntax:

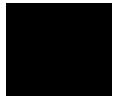
Label	Operation	Operand	Comment
	RESTORE		

#### Description:

The RESTORE directive restores those options that were previously saved by the SAVE command. Once RESTORE is specified, all options specified after the last SAVE will no longer have any effect.

#### Example:

```
OPT P=68010
. . .
SAVE
OPT P=68020 ; 68020 instructions are now legal
. . .
CAS D0,D1,(A3) ; 68020 instruction
. . .
RESTORE
. . . ; 68020 instructions are no longer legal
. . .
END
```



## **SAVE**

### **Save Assembler Options**

**Syntax:**

Label	Operation	Operand	Comment
	SAVE		

**Description:**

The SAVE directive saves the current set of OPT options (see the OPT command for a list of these options). The processor and coprocessor types are also saved.

The options can be restored at a later time with the RESTORE command. Once RESTORE is specified, all options specified after the last SAVE will no longer have any effect.

**Example:**

```
OPT P=68010
. . .
SAVE
OPT P=68020 ; 68020 instructions are now legal
. . .
CAS D0,D1,(A3) ; 68020 instruction
. . .
RESTORE
. . . ; 68020 instructions are no longer legal
. . .
END
```

## SECT, SECTION

### Specify Section

**Syntax:** There are three distinct syntaxes:

Label	Operation	Operand	Comment
{label}	SECT{.S}	sname{,align}{,contents}{,HPtype}	
	SECT{.S}	snumber{,align}{,contents}{,HPtype}	
label	SECT{.S}	snumber{,align}{,contents}{,HPtype}	

**Where:**

label Specifies a label equal to the address of the current program counter. If snumber is specified, label cannot be used if SECT is a non-common section. If snumber is a common section, then snumber and label will be combined to form the section name.

.S Assigns the absolute short attribute to the section. All symbols specified will be found in an area of memory accessible by a 16-bit address, or they will be constants with a 16 bit or smaller value.

sname Symbol name. Any valid symbol may be used.

align The number of bytes of alignment, 1, 2 or 4. The section alignment attribute allows you to specify that a section be located on a 1, 2, or 4 byte boundary. Refer to the "Relocation" chapter for more information.

contents Contents of section:

- C - Program code.
- D - Data.
- M - Mixed code & data.
- R - ROMable Data.

The section contents attribute is used by HP debuggers to gain efficiency. See *Relocation* chapter.

Assembler Directives  
**SECT, SECTION**

HPtype            Specifies how to map this section on to the HP 64000  
 asmb\_sym and link\_sym files

- A - ABS
- C - COMN
- D - DATA
- P - PROG

number            Section number. Up to two decimal digits may be used.

**Description:**

The SECT directive specifies to the assembler that the following statements should be assembled in the relocatable section specified, which remains in effect until an ORG, OFFSET, COMMON or another SECT or SECTION directive is assembled that specifies a different section. Initially all section location counters are set to zero.

SECT and SECTION are completely equivalent.

The user may alternate between the various sections with multiple section directives within one program. The assembler will maintain the current value of the location counter for each section.

Creating a common section name by combining the label and section number is not a behavior that is consistent with the Motorola assembler.

**Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
LABEL1	SECT	SECT1	;name is SECT1, LABEL1 is ;normal symbol.
	SECT.S	CODE	;name is CODE.
	SECTION	0	;name is 0, noncommon section.
LABEL1	SECTION	0	;name is 0LABEL1, common section.
	SECT	A,4	;in each file, 1st byte of ;section A is quad aligned.
	SECT	B,4,C	;quad aligned, section ;type = program code.
	SECT	C,,D,C	;C section type = data. ;HP 64000 section COMN



## SET

### Equate a Symbol to an Expression

**Syntax:**

Label	Operation	Operand	Comment
label	SET	expression	

**Where:**

label	A symbol defined by this statement.
expression	A value that will be assigned to the given label until changed by another SET directive. Any symbols used in the expression must be previously defined.

**Description:**

The SET directive sets a symbol equal to a particular value. Unlike the EQU directive, multiple SET directives for the same symbol may be placed in a source program. The most recent SET directive determines the value of the symbol until another SET directive is processed.

Like EQU, this directive may also be used to define new keywords.

**Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
GO	SET	5	
GO	SET	GO+10	

## **SPC**

### **Space Lines on Listing**

**Syntax:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	SPC	expression	

**Where:**

expression      Must evaluate to an absolute value that determines how many lines are to be skipped. It may not be relocatable, but may contain forward references.

**Description:**

This directive causes one or more blank lines to appear on the output listing. It enables the programmer to format the listing for easier reading. The directive itself does not appear in the listing.

The user may also use a blank source statement to insert blank lines on the listing.

**Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	SPC	7	

## TTL

### Set Program Heading

**Syntax:**

Label	Operation	Operand	Comment
	TTL	heading	

**Where:**

heading	Title that will be placed at the beginning of each page. Up to 60 characters may be used in the heading, with additional characters being ignored. The heading may optionally be delimited by single quotes, as shown in the example. If so, the quotes are not considered part of the title. If the terminating quote is not present, the first 60 characters will be used.
---------	--

**Description:**

This directive is used to print a heading at the beginning of each page of the listing, in addition to the line identifying the listing as output of the as68k assembler. The default heading defined by the assembler is all blank. For a user specified title to appear on the first page of the output listing, this directive must be the first statement in the program.

**Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	TTL	'TEST PROGRAM'	

## **XCOM**

### **Specify Weak External References**

**Syntax:**

Label	Operation	Operand	Comment
	XCOM	symbol,size	

**Where:**

symbol	The name of a symbol referenced in this module but defined in a different module or by the linker.
size	The size in bytes that the linker will reserve if there is no specific public definition for this symbol.

**Description:**

The XCOM directive specifies a symbol that is referenced in this module but is assumed to be defined in a separate module. If no module defines symbols, then the linker will reserve space for the symbol in a section named "zerovars".

This directive was created to support the assembly of compiler-generated assembly code. Some languages like ANSI C permit several modules to define the same variable. In order to prevent duplicate symbol errors, a compiler might generate XCOM directives for its variables instead of defining variables in each module. The linker will then allocate space for the symbols.

XCOM directives can appear anywhere within the program. You can declare common symbols to be externally defined multiple times. Common symbol references can appear in any section including absolute sections.

A size (in bytes) must be supplied so that if the linker must define the symbol, the appropriate size will be allotted.

**Example:**

```
XCOM PROC1,1
```

In this example, the weak external reference for the symbol PROC1 assumes that the final value is long. If the linker must define its value, one byte of space will be reserved for it.

## XDEF

### Specify External Definition

**Syntax:**

Label	Operation	Operand	Comment
	XDEF	symbol list	

**Where:**

symbol list      A list of symbols separated by commas that specify the names defined in this module and to be referenced by other modules. Symbols in the list cannot be separated by spaces.

**Description:**

This directive specifies a list of symbols that will be given the external definition attribute. These symbols will then be made available to other modules by the linker. Symbols appearing in this directive are placed in the object module.

XDEF may appear anywhere within the program and each symbol may be declared multiple times. Declarations after the first, for any given symbol, will be ignored.

Symbols that are declared with this directive (but not defined in the program) will be flagged as undefined in the output listing.

**Note**

An XDEF will override a previous XREF for any symbol that has not been previously defined.

**Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	XDEF	SCAN, LABEL, COSINE	

## **XREF**

### **Specify External Reference**

**Syntax:**

Label	Operation	Operand	Comment
	XREF{.S}	{sname:}symbol{,{sname:}symbol,...}	

**Where:**

.S	Means that all symbols in the statement will be found in the area of memory accessible by the absolute short mode, or will be constants with 16-bit or smaller values. Also, any section whose name or number appears in the statement is designated as short.
sname	A section name or number.
symbol	The name of a symbol referenced in this module that is defined in a different module. Spaces are not allowed between the symbols in the list.

**Description:**

This directive specifies a list of symbols that will be given the external reference attribute, and optionally assigns to each symbol the name (or number) of a relocatable section. External references are symbols that are referenced in this program module but defined within another program. The XREF directive provides the linkage to those symbols through the Linking Loader.

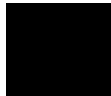
XREF may appear anywhere within the program and each symbol may be declared multiple times. Declarations after the first, for any given symbol, will be ignored.

Specifying the section name (or number) of an external reference sometimes affects the assembler's choice of address mode (refer to the "Instructions and Address Modes" chapter). Also, during the Linking process, the Loader will verify that the externally referenced symbol is indeed in the specified section. An external reference with no section name or number specified is presumed to be absolute for the purpose of selecting addressing modes.

A section name (or number) applies to all symbols following it, until the appearance of another section name (or number) or the end of the statement. It is legal for a section name to appear only in XREF statements. In this case, however, it counts toward the total of 200 allowable section names.

**Example:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	XREF	PROC1, PROC2, SECT1	INPUT, 2:OUTPUT



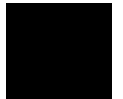
Assembler Directives  
**XREF**





---

# 7



---

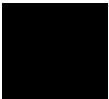
## Macros

This chapter defines the parts of a macro and describes some directives you can use to define macros.

## Macros

A macro is a sequence of instructions that can be automatically inserted in the assembly source text by encoding a single instruction, known as the macro call. The macro definition is written only once, but can be called any number of times. The macro definition may contain parameters which can be changed for each call. The macro facility simplifies the coding of programs, reduces the chance of programmer error, and makes programs easier to understand, since the source code need only be changed in one location, which is in the macro definition.

A macro definition consists of three parts: a heading, a body, and a terminator and must precede any call to this macro. A macro may be redefined at any place in the program, but the most recent definition of a macro name will be used when the macro is called. A standard assembler mnemonic (e.g., OR) may also be redefined by defining a macro with the name OR. In this case all subsequent uses of the OR instruction in the program will cause the macro to be expanded.



---

## Macro Heading

The heading, which consists of the directive MACRO, gives the macro a name and defines any formal parameters.

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
label	MACRO	{parameter list}	

Label specifies the macro name and must not end and must not contain a period (such as for as size qualifier like .W) because these strings will be interpreted as a qualifier or a shorter name when the macro is invoked later, and the correct macro will not be referenced. Other than this, the macro name may be any legal symbol and it may be the same as other program defined symbols since it has meaning only in the operation field. For example, TAB could be the name of a symbol as well as a macro.

If a macro name is identical to a machine instruction or an assembler directive, the mnemonic is redefined by the macro. Once a mnemonic has been redefined as a macro, there is no way of returning that name to be a standard instruction mnemonic. A macro name may also be redefined as a new macro with a new body.

The operand field of the MACRO line may contain the names of dummy formal parameters in the order in which they will occur on the macro call. Each parameter is a symbol and multiple parameters must be separated by commas. The symbols used as formal parameters are known only to the macro definition and may be used as regular symbols outside the macro.

Named formal parameters need not be specified. Unnamed parameters (and named parameters as well) can be referenced with the Motorola backslash notation (described below) in terms of the parameter's position on the call line. However, unnamed (i.e., null) formal parameters are not allowed if they are followed by any named parameters; for example, "XYZ MACRO „PARAM3" is not allowed. This means that unnamed parameters must either come after all named parameters on the macro definition line or must be assigned a dummy name.

## Macro Body

The first line of code following the MACRO directive that is not a LOCAL directive is the start of the macro body. MACRO body statements are placed in a macro file for use when the macro is called. During a macro call, an error will be generated if another macro is defined within a macro. No statements in a macro definition are assembled at definition time; they are simply stored in the macro file until called, at which time they are inserted in the source code at the position of the macro call.

The name of a formal parameter specified on the MACRO directive may appear within the macro body in any field. If a parameter exists, it is marked, and the real corresponding parameter from the macro call will be substituted when the macro is called. Parameters are not recognized in a comment statement or in the comment field of a statement, provided the comment field is prefixed by a semicolon (;).

Alternatively, parameters may be referenced in the form \n where n is a non-negative integer. Parameter \0 is the qualifier (extension) of the macro call and may appear only as a qualifier on opcodes in the macro body. (This is the only format in which this qualifier can be referenced). Parameters \1,\2...\9,\A,...\Z are the first,second... real parameters on the macro call line.

Macro parameters will be expanded in a quoted string. But, if the quoted string is preceded by "A" or "E" (for ASCII or EBCDIC), macro parameters are not recognized within the string. This extension permits backslashes and formal parameter names to appear as a string when the user so desires.

When referring to macro parameters in the macro body, you may precede the macro parameter with "&&". This allows you to embed the parameter in a string. For example:

```
1          MAC1   MACRO   P1
2          L&&P1   MOVE   D0,D1   ; Create label using parameter.
3          ENDM
4
5          MAC1   XX      ; Call macro.
5.1 00000000 3200  LXX    MOVE   D0,D1   ; Create label using parameter.
6          END
```

## Macro Terminator

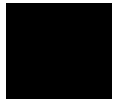
The ENDM directive terminates the macro definition. During a Macro definition an ENDM must be found before another MACRO directive may be used. An END directive also terminates a macro definition as well as the assembly of the file in which it is contained.

The format of the ENDM directive is as follows:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
{label}	ENDM		

### Where:

label                      An optional label that becomes the symbolic address of the first byte of memory following the inserted macro. Labels with embedded parameters are not allowed on the same line as the ENDM directive. The label can be placed on the line preceding the ENDM directive for the desired effect.



## Macro Call

A macro may be called by encoding the macro name in the operation field of the statement.

The format of the call is shown below.

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
{label}	name	{.qualifier} {parameter list}	

### Where:

label	An optional label that will be assigned a value equal to the current program counter.
name	The name of the macro called. This name should have been defined by the MACRO directive or an error message will be generated.
qualifier	An optional qualifier that may be B, W, L or S and is passed to the macro as parameter \0.
parameter	A list of parameters separated by commas. Parameters may be constants, symbols, expressions, character strings or any other text separated by commas. The number of parameters cannot exceed 35.

The parameters in the macro call are actual parameters and their names may be different than the formal parameters used in the macro definition. The actual parameters will be substituted for the formal parameters in the order in which they are written. Commas may be used to reserve a parameter position. In this case, the parameter will be null (i.e., contain no actual characters). The formal parameter corresponding to a null actual parameter is simply removed during macro expansion. Any parameters not specified will be null. The parameter list is terminated by a blank, tab, newline, or semicolon. The macro processor does not recognize a semicolon as a delimiter. A comment beginning with a semicolon following the parameter list must be separated from the parameter list by a blank or tab (white space).

All actual parameters are passed as character strings into the macro definition statements. Thus, symbols are passed by name and not by value. In other words, if a symbol's value is changed in the macro, in its expansion it will also have the new value outside of the macro. Thus SET directives within a macro body may alter the value of parameters passed to the macro.

The angle brackets (< >), are used to delimit actual parameters that may contain other delimiters. When the left bracket is the first character of any parameter, all characters between it and the matching right bracket are considered part of that parameter. The outer brackets are removed when the parameter is substituted in a line. Angle brackets may be nested for use within nested macro calls. **The brackets are the ONLY way to pass a parameter that contains a blank, comma, or other delimiter.** For example, to use the instruction "ROL # 1,D1" as an actual parameter would require placing < ROL # 1,D1> in the actual parameter list. A null parameter may consist of the angle brackets with no intervening characters, but the characters < and > may not be passed as parameters and the parameter \0 may not contain angle brackets.

An example of a macro call and its expansion is shown below. Note that expanded code is marked with plus signs.

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
GET	MACRO	W,Y,Z	;macro definition
	MOVE	#w,D5	
	ROL	#1,D5	
	Y		
z	JMP	\4	
	ADD.\0	#5,D0	
	ENDM		
	-		
LOOP	GET.B	200,<BRA DATA>,ENTRY,MAIN	;macro call
	JMP	FIRST	
	-		
	-		
LOOP	GET.B	200,<BRA DATA>,ENTRY,MAIN	;macro expansion
+	MOVE	#200,D5	
+	ROL	#1,D5	
+	BRA	DATA	
+ENTRY	JMP	MAIN	
+	ADD.B	#5,D0	
	JMP	FIRST	

The operator double equal sign ( == ), pronounced "exists", may be used to determine whether a parameter is present or not in the macro call. This operator returns a true value (all ones) if any operand follows the == and a

## Macros

### Macro Call

false value (all zeros) otherwise. For example, the following code checks whether the second parameter is present.

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
MSET	MACRO	DATA, MEM	
	IFNE	==MEM	
	MOVE	#DATA, MEM	
	ELSEC		
	MOVE	#DATA, (A1)	
	ENDM		

The == operator may be used in combination with other operators. It takes as its argument the entire remainder of the line, up to a comment delimiter (if present) or the end of the line. Therefore, using other operators to the right of == is useless. Also, if a comment field is to follow an == operator, it must be prefixed with a semicolon (;). A parameter consisting entirely of blank characters will test as null.



## LOCAL - Define Local Symbol

All labels, including those within macros, are known to the entire program. A macro containing a label that is called more than once will cause a duplicate label error to be generated. To avoid this problem, the user may declare labels within macros to be local to the macro. Each time the macro is called the assembler assigns each local symbol a system generated unique symbol of the form ??nnnn. Thus, the first local symbol will be ??0001, the second ??0002, etc. The assembler does not start at ??0001 for each macro, but increases the count for each local symbol encountered. The maximum number of local symbols allowed inside a macro definition is 90.

The symbols defined in this directive are treated like formal macro parameters and hence may be used in the operand field of instructions. The operand field of the LOCAL directive may not contain any formal parameters defined on the MACRO directive line. As many LOCAL directives as necessary may be included within a macro definition but they must occur immediately after the MACRO directive and before the first line of the macro body, including comment lines. LOCAL directives that appear outside a macro definition will generate an error.

For compatibility with existing code, the assembler will also recognize the Motorola method of declaring local symbols. The string "@" denotes the presence of a local symbol. The full name of the symbol is formed by concatenating "@" with any adjacent symbol(s) (e.g., "DON@T" counts as one local symbol). The total length of a symbol formed in this way should not exceed 31 characters, or the assembler may not resolve it correctly. At macro expansion time, the entire local symbol is replaced by a symbol of the form ??nnnn, just like named local symbols. This form may be mixed with named local symbols without conflict (although this is not recommended).

Local symbols declared by the "@" construction may not be present in a LOCAL statement, but are recognized as they appear.

The \@ format is not recommended for new code, as it obscures the meaning of the macro definition without adding clarity to the expansion.

Macros  
**LOCAL - Define Local Symbol**

**Syntax:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	LOCAL	symbol list	

**Where:**

symbol list                      A list of symbols that are separated by commas and that are to be defined local to this macro.

Example of local symbol usage:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
WAIT	MACRO	TIME	;macro definition
	LOCAL	LAB1	
LAB2\@	MOVE.B	#TIME,D0	
LAB1	DBLE	D0,LAB2\@	
	ENDM		
??0002	MOVE.B	#5,D0	;First call
??0001	DBLE	D0,??0002	;with TIME=5.
??0004	MOVE.B	#\$FF,D0	;Second call
??0003	DBLE	D0,??0004	;with TIME=\$FF

## MEXIT - Alternate Macro Exit

The MEXIT directive provides an alternate method for terminating a macro expansion. During a macro expansion, an MEXIT directive causes expansion of the current macro to stop and all code between the MEXIT and the ENDM for this macro to be ignored. If macros are nested, MEXIT causes code generation to return to the previous level of macro expansion. Note that either MEXIT or ENDM may be used to terminate a macro expansion, but only ENDM may be used to terminate a macro definition.

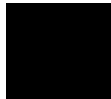
### Syntax:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
{label}	MEXIT		

### Where:

label	An optional label that will be given the address of the current location counter. In the following example, the code following the MEXIT will not be assembled if DATA is non-zero.
-------	---

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
STORE	MACRO	DATA	
	-		
	-		
	IFEQ	DATA	
	MEXIT		
	ENDC		
	-		
	-		
	ENDM		



## Macro Parameter Count

The special symbol NARG may be used when it is necessary to know the number of parameters passed on the macro call statement to the macro. This symbol is used like any other symbol and represents the number of actual parameters passed to the macro, as opposed to the number of formal parameters in the macro definition. NARG is considered to be zero outside of a macro. It is typically used when generating tables within macros, along with conditional assembly statements. This count only represents parameters that are not null.

Example:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
GEN	MACRO	P1,P2,P3	
	IFNE	NARG	
	DC.B	P1,NARG	
	GEN	P2,P3	
	ENDC		
	ENDM		
ADD1	EQU	\$7F	;macro call.
ADD2	EQU	3	
	GEN	ADD1,ADD2	
* Macro Expansion:			
	IFNE	NARG	;(value of NARG)
7F02	DC.B	ADD1,NARG	
	GEN	ADD2,	
	IFNE	NARG	
0301	DC.B	ADD2,NARG	
	GEN	,	
	IF	NARG	
	DC.B	,NARG	; not executed
	GEN	,	; not executed
	ENDC		
	ENDC		
	ENDC		

Note that the value of NARG is not displayed in the expansion, any more than the value of any other symbol. In the example above the DC.B directive is used so that the value of NARG can be seen.

---

## Structured Control Statements

This chapter describes the high-level control directives which you may use in your assembly language programs.

## Structured Control Statements

as68k includes several high level language constructs, like those of C and Pascal, that control runtime loops and conditional execution. These constructs make it easier to write fast, compact assembly language code. The following control directives are provided:

- IF ... ELSE ... ENDI
- WHILE ... ENDW
- REPEAT ... UNTIL
- FOR ... ENDF

Within the constructs, the following keywords may also be used:

THEN, DO, TO, DOWNTO, AND, OR, and BY.

The following extensions to the Motorola control directives alter the flow of the loop constructs:

- BREAK
- NEXT

BREAK may be used to prematurely exit a loop. NEXT may be used to proceed to the next iteration of the loop.

Each of the structured control directives generates one or more assembly language instructions. The instructions generated typically include compare and branch instructions.

Operands give you control over which registers and memory locations are used to hold the loop counts or values to be compared for the loop end conditions. There is no restriction on storing into the loop counter, loop increment variable, or either of the loop bounds for the loop. When writing code for the loop body, be careful not to alter these variables.

The IF structure directive should not be confused with the IFxx conditional assembly directive. At assembly time, each structure directive is translated into the appropriate assembly language code that will be executed at run time. Conditional assembly directives do not generate any code; they only control what will and will not be assembled.

## Structured Control Expressions

The IF, UNTIL and WHILE statements require a field referred to as a "structured-control expression" in their syntax. This expression has a logical value of "true" or "false" and is one of the following:

- 1 A condition code (CC, EQ, etc.) enclosed in angle brackets. For example: "<MI> ". Any of the 14 condition codes accepted in the conditional branch instruction (Bcc) is legal.
- 2 Two expressions as defined in the "Expressions" section of the "Assembler Syntax" chapter, separated by a condition code enclosed in angle brackets (e.g., "COUNT <LE> # 4"). These expressions will be used as operands for the CMP instruction; if they do not form a legal pair of operands for this instruction, an error will occur when the CMP is assembled. The # sign is required on all immediate operands, as in the example.
- 3 Two structured-control expressions, each of either type 1 or type 2 above, separated by the keywords AND or OR. These keywords may optionally have one of the qualifiers .B, .W or .L (e.g., COUNT <LE> # 4 AND.B <CC> ").

More complex combinations, such as "COUNT <LE> # 4 AND <CC> OR X <GT> Y", are not allowed. As in the examples, at least one space or tab must appear between different parts of a structured-control expression.

The first type of structured-control expression generates a conditional branch instruction (Bcc), which merely tests the indicated bits of the condition codes. (The test may be complemented to reflect the programmer's intent in some constructs.) Obviously, these codes should somehow be previously set. The expression is "true" if the condition code setting described is true.

The second type of structured-control expression generates a CMP (compare) instruction followed by a conditional branch. The size of the CMP is controlled by the qualifier on the directive containing the structured-control expression. It is not always possible to produce a single conditional branch that is equivalent in meaning to the expression coded; this is further discussed below.

The third type of structured-control expression generates the code for its left side followed by the code for its right side: there are no extra instructions generated by the AND or OR. The branches are constructed so that the right side of AND is not evaluated when the left side is false (the compound

## Structured Control Statements

### Structured Control Expressions

expression is known to be false), nor is the right side of OR evaluated when the left side is true (the compound expression is known to be true). The size of the CMP (if any) to the left of the AND or OR is taken from the qualifier on the directive; the size of the CMP (if any) to the right of the AND or OR is taken from the qualifier on the AND or OR. A compound expression containing AND is true if and only if the expressions on both sides of AND are true, otherwise it is false. A compound expression containing OR is false if and only if the expressions on both sides of OR are false, otherwise it is true.

The assembler normally uses the expression preceding a condition code as the left operand of CMP, and the expression following the condition code as the right operand of CMP. But if this is not a legal combination of operands for CMP, the assembler will switch the operands and leave the specified condition code alone. To preserve the meaning of the specified comparison, the assembler will change the condition code as follows.

<CC>	<==>	<LS>	
<CS>	<==>	<HI>	
<EQ>	<==>	<EQ>	
<NE>	<==>	<NE>	
<GE>	<==>	<LE>	
<GT>	<==>	<LT>	
<PL>	<==>	<MI>	*
<VC>	<==>	<VC>	*
<VS>	<==>	<VS>	*

In the first six cases, the new condition is exactly equivalent. In the last three (asterisked), it is not always and is marked with a warning message flag on the assembly listing when it occurs. The conversions of VC to VC, and VS to VS, fail when the result of the comparison is the largest negative number representable in the operation size (\$80, \$8000, or \$80000000). The conversion of PL to MI or of MI to PL fails in the same case, and also when the result of the comparison is 0. It is recommended that such flagged expressions be recoded to express the programmer's intent.



---

## FOR...ENDF Loop

### Syntax:

```
FOR{.qualifier} op1 = op2 TO op3 {BY op4} DO{.extent}
    <loop body>
ENDF
```

or:

```
FOR{.qualifier} op1 = op2 DOWNTO op3 {BY op4} DO{.extent}
    <loop body>
ENDF
```

These statements are iterated loops, like the FOR of Pascal or C and the DO of FORTRAN. The loop counter is "op1", which must be an expression that is legal as the right side of a MOVE instruction (typically a label or a register). The initial value is "op2" and "op3" is the final value of this counter. On each pass through the loop, "op1" is incremented for TO (decremented for DOWNTO) by "op4" if present, or by 1 if "op4" is not present. The loop is executed until "op1" is greater than "op3" for TO ("op1" less than "op3" for DOWNTO), which means that it may be executed zero times if "op1" is greater than "op3" (for TO) when the loop is entered.

The loop body may be any statements, but if any structured control statements are included, they must be nested properly.

The FOR...ENDF loop generates a MOVE, a CMP, and either an ADD or SUB, plus various conditional and unconditional branches. The MOVE, CMP and ADD or SUB may all have a qualifier that is taken from the qualifier field of the FOR statement for all three instructions. The CMP is performed at the top of the loop, which means that the following conditional branch out of the loop is a forward reference. This branch may be given an explicit size code (.S or .L) by appending the code to the DO keyword as the "extent" field. If not present, the size of the forward branch is determined by the current setting of the B option (OPT BRL or OPT BRS).

The generated CMP instruction is executed once, even if the values of "op1" and "op3" are such that the body of the loop is executed zero times. Upon exit from the loop, "op1" will contain the last value to which it was incremented/decremented (which will be outside the range of the loop bounds) and the condition codes will reflect the failing CMP. Unlike most

## Structured Control Statements

### FOR...ENDF Loop

high-level languages, there is no restriction on storing into the loop counter, loop increment, or either of the loop bounds within the loop (of course doing this is error prone).

Spaces or tabs are required as separators as shown above. **Note particularly the required spaces around the equals sign.**

Fields "op1" through "op4" are used as instruction operands just as they appear; if a legal instruction is not produced, errors will occur when the generated instruction is assembled. Any immediate data must have # signs attached. If any operand is an A register, the qualifier on FOR must not be .B (byte). The default increment size of 1 is usually inappropriate when branching through Word or Long sized data.

### Examples:

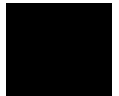
<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	FOR.B D1 =	#1 TO #10 DO.S	
		MOVE.W D1, (A2)+	
	ENDF		
	FOR.L A1 =	#HIGHADD DOWNT0 #LOWADD BY #4 DO	
		MOVE.L (A1), -(A2)	
	ENDF		

## IF ... THEN ... ELSE ... ENDI Conditional Execution

### Syntax:

```
IF{.qualifier} <structured-control-expression> THEN{.extent}  
    <then-part>  
{ELSE{.extent}  
    <else-part> }  
ENDI
```

This means that only the statements in the then-part are to be executed if the < structured-control-expression> is true, and only the statements in the (optional) else-part are to be executed if the < structured-control-expression> is false. The qualifier on IF is used when generating code for the < structured-control-expression> as explained above. The extent code on THEN, which may be .S or .L, is used when generating the conditional branch from the test (at IF) to the else-part. Similarly, the extent code on ELSE is used when generating the unconditional branch from the end of the then-part to the end of the else-part.



Structured Control Statements  
**IF ... THEN ... ELSE ... ENDI Conditional Execution**

**Examples:**

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	IF.B (A1) <LT> #0 THEN.S		
		MOVE.B #0,(A1)	
	ELSE.S		
		ADD.B #1,(A1)	
	ENDI		

;This example shows mixed conditional assembly and structured  
 ;syntax IFs.  
 ;As you see, the combination is difficult to understand  
 ;sometimes.

IFNE VARIABLE			;conditional
	IF VARIABLE <NE> #0 THEN.S		;structured
		MOVE #0,VARIABLE	
	ELSE.S		;unambiguously structured
			;because of .S, no W flag is
			;given
		JSR ERROR	
ELSEC			;conditional, because
			;structured is illegal
	IF VARIABLE <EQ> #0 THEN.S		;structured
		MOVE #1,VARIABLE	
ENDC			;conditional
	ENDI		;structured- terminates
			;whichever of the preceding
			;structured IF's was assembled

## REPEAT ... UNTIL Loop

### Syntax:

```
REPEAT      <loop body>
UNTIL{.qualifier} <structured-control-expression>
```

The loop is executed until the < structured-control-expression> becomes true. The test is placed at the end of the loop, so that the loop body is executed once, even if the < structured-control-expression> is true upon entry to the loop.

The REPEAT generates only a label and UNTIL generates code for the < structured-control-expression> as described above. Since all branches involved are backwards, there is no need for an extent field. The qualifier of UNTIL is used in generating code for the < structured-control-expression>, as explained earlier in the previous "STRUCTURED-CONTROL EXPRESSIONS" section of this chapter. A comment field on UNTIL must be delimited by a semicolon or exclamation point, so that the assembler will know to stop parsing the < structured-control-expression> .

### Examples:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	REPEAT		
		MOVE.L #-1,(A1)+	
		MOVE.L #0,(A1)+	
	UNTIL.L A1	<GE> #FF8000	
		ANDI #\$FE,CCR	; clear Carry flag
	REPEAT		; this infinite loop might be used
	UNTIL	<CS>	; while awaiting an external interrupt



## WHILE ... ENDW Loop

### Syntax:

```
WHILE{.qualifier} <structured-control-expression> DO{.extent}  
    <loop body>  
ENDW
```

This means to repeat the < loop body> provided that the < structured-control-expression> remains true. If it is false upon loop entry, then the loop body is executed zero times (but the CMP test is executed once and the condition codes will reflect this).

The qualifier on WHILE is used when generating code for the < structured-control-expression> as explained above. The extent field of the DO is applied to the conditional branch from the test out of the loop, which is a forward reference.

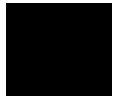
### Examples:

<u>Label</u>	<u>Operation</u>	<u>Operand</u>	<u>Comment</u>
	WHILE A1 <NE>	#0 DO.S	
		MOVE #0,(A1)+	
	ENDW		
	WHILE.L #3 <LT>	D0 AND.L #5 <LT>	D1 DO.S
		JSR RETRY	
	IF.L #5 <LT>	D1 THEN.S	
		ADD.L #1,D1	
	ELSE.S		
		MOVE.L #0,D1	
		ADD.L #1,D0	
	ENDI		
	ENDW		

## **BREAK - Premature Loop Exit**

The BREAK directive provides a convenient way to exit a loop (FOR, WHILE or REPEAT) before the condition terminating the loop becomes true. BREAK generates a jump to the assembler-generated label (which you do not know when coding the program) that comes immediately after the innermost active loop in which the BREAK appears. Since this branch is a forward reference, an extent code .S or .L may be attached to the BREAK directive to force either a short or long forward branch.

If a BREAK directive appears outside of a FOR-ENDF, WHILE-ENDW, or REPEAT-UNTIL loop, an opcode error is reported and no code is generated. **BREAK is not allowed in an IF construct.**



## **NEXT - Proceed to Next Loop Iteration**

The NEXT directive provides a convenient way to proceed to the next iteration of a loop (FOR, WHILE or REPEAT). NEXT generates a jump to the assembler-generated label at the bottom of the innermost active loop in which the NEXT directive appears. Since this branch is a forward reference, an extent code .S or .L may be attached to the NEXT directive to force either a short or long forward branch.

If a NEXT directive appears outside of a FOR-ENDF, WHILE-ENDW, or REPEAT-UNTIL loop, an opcode error is reported and no code is generated. **NEXT is not allowed in an IF construct.**



## Structured Directive Nesting

Structured directives may be nested to create multi-level control structures subject to the following rule. A directive that begins a new control structure in an inner loop must have a corresponding directive that terminates the control structure in the same inner loop.

The assembler keeps track of structured control directives to ensure that they are nested properly. The maximum nesting level is 64. This process is totally independent of the assembly time macro stack and conditional assembly stack. It is possible for the beginning of a structured control loop to be inside a conditional assembly or a macro expansion. The directive ending the structured control loop must be specified, but it need not be within the conditional assembly or macro expansion.

An incorrectly nested control directive is flagged with an invalid opcode error and ignored by the assembler. If a terminating directive is omitted, an undefined label error will follow the control directive beginning the high level construct.

An example of legal nesting is shown in the following example:

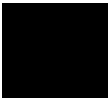
```

REPEAT
  MOVE.B (A1)+,NEXT_CHAR      ; Fetch character.
  CMP.B #CR,NEXT_CHAR        ; We cannot use IF here because
  BNE.S label1               ; BREAK cannot be nested in an
                              ; IF structure.
                              ; Without the BREAK, we could use
                              ; IF.B #CR <EQ> NEXT_CHAR THEN.S
  BREAK.S                    ; Leave the REPEAT...UNTIL loop
                              ; when carriage return is found.
label1
  IF.B #BLANK <EQ> NEXT_CHAR THEN.S ; Skip blanks.
  BRA.S label2               ; Cannot use NEXT in an IF.
  ELSE.S
  MOVE.B NEXT_CHAR,(A2)+      ; Copy character into buffer.
  IF.L A2 <GT> #120 THEN.S    ; Error if buffer overflows.
  JSR ERROR
  ENDI
  ENDI
label2
  UNTIL A1 <GT> #120
  RTS
END

```

## **Structured Directive Listings**

The code generated by structured control directives is shown in the same way on the listing as macro expansions. The code is marked with plus signs (+), and is not shown if the M or MEX option is turned off.



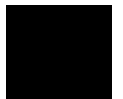
---

# 9

---

## Linker/Loader Introduction

This chapter and subsequent chapters describe the 68000 Family Linking Loader that accompanies the 68000 Family Relocatable Macro Assembler.



## Linker/Loader Introduction

The linking loader may be used to combine several independently assembled relocatable object modules into a single absolute object module. Relocatable addresses are transformed into absolute addresses, external references between modules are resolved, and the final absolute symbol value is substituted for each relocatable symbol reference.

In addition, ld68k supports incremental linking. In an incremental link, several relocatable modules are combined into a single relocatable file that may be used in a subsequent linking operation. The output file format is HP's implementation of IEEE standard 695.

During incremental links, location information may be specified but, because the code remains in relocatable form, these locations may be changed during subsequent links. A number of linker commands are illegal in an incremental link.

## **Linker/Loader Features**

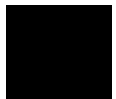
The 68000 family linking loader supports the following features:

- All relocatable section load addresses may be specified independently.
- The relocatable section loading order may be specified.
- External symbols may be defined or the values of previously defined externals may be changed at load time.
- Object modules may be loaded from a library created by the librarian.
- Symbols and linenumber information may be included in the absolute object module for symbolic debugging.
- A cross reference table of external symbols and all modules in which they are referenced may be generated.
- First fit memory may be allocated for more compact load modules.
- Complex relocation is supported.
- A2 - A5 Relative addressing is provided.
- Relocatables may be combined into a single relocatable in a process called incremental linking.
- Data initialization from ROM is supported.
- Multiple address spaces are supported.

---

## **Linker/Loader Operation**

To avoid long assembly times, or to reduce the required size of the assembler symbol table, long programs can be subdivided into smaller modules, assembled separately, and linked together by the loader program. After the separate program modules are linked and loaded, the output module functions as if it had been generated by a single absolute assembly.



## Linker/Loader Introduction

### Program Sections

The same program does the linking and loading for the 68000 Family Cross Assembler/Linker/Librarian. The names "loader," "linker," "linking loader," and "linker/loader" all refer to the same program. This chapter will use the name "loader."

The primary functions of the loader are to:

- Resolve external references between modules and check for undefined references. (The linking process.)
- Adjust all relocatable addresses to the proper absolute addresses. (The loading process.)
- Output the final absolute object module(s).

---

## Program Sections

To use the assembler and loader effectively, you should understand the various program sections and section load addresses.

### Absolute Section

This section is that part of the assembly program that is not relocatable but is to be loaded at fixed locations in memory. Absolute code is placed into the output module exactly where specified by the input object modules. If no code is generated by an instruction (the DS directive, for example), no code is placed into the output module.

### Relocatable Section

A relocatable section is a general purpose section which may contain both instructions and data. A program may contain an unlimited number of relocatable sections.

Each section is identified by a symbolic name. The same section name may appear in different relocatable object modules. The section, as a whole, refers to the totality of code from all object modules which is associated with the section name. Instructions in one section can make reference to any other section.

In the assembler, sections may be given numbers rather than names. If a label appears before a SECTION directive which defines a numbered section, the assembler creates a section name made up of the number and the label.

However, from the loader's point of view, all sections are named.

On occasion it will be necessary to refer to the individual pieces of code from various modules which make up a section; these will be called subsections.

Each relocatable section has five attributes: the common/noncommon attribute, the short/long attribute, the alignment attribute, the section contents attribute, and the HP type attribute.

### **Noncommon Section**

A noncommon section is the only type available for code. The subsections of a noncommon section are loaded into a contiguous block of memory and do not overlap. The size of a noncommon section is the sum of the sizes of all its subsections.

### **Common Section**

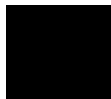
These sections contain variables that may be referenced by each module. All common subsections are loaded beginning at the same address providing an effective communication area. This is similar to FORTRAN Common.

The length of a common section is the size of its largest subsection. If more than one input subsection contains code or data in the same Common section, the linker will issue a warning.

### **Short Section**

A section which may be referenced by the absolute short address mode and which therefore must be loaded into the areas of memory which can be reached by a 16-bit sign-extended address. These areas are from 0 to \$7FFF inclusive, plus another area in high memory whose boundaries depend on the target chip. For the 68000 and 68010 this area is from \$FF8000 to \$FFFFFF inclusive; for the 68008 it is from \$F8000 to \$FFFFFF inclusive; for the 68020/30/40 and CPU32, it is from \$FFFF8000 to \$FFFFFFFF inclusive. The target chip may be specified by the CHIP command.

The loader never puts a short section in an inappropriate area of memory. A section is designated as short if any of its subsections are short, or if it appears



in a SORDER directive in the loader commands. A target system may choose not to implement all the available address lines for the target microprocessor. For example, the 68020 has 32 address lines, but perhaps the target system uses only 24 to control memory. In this case, the loader CHIP command may be used to specify a bus width of 24 lines and therefore a target memory less than  $2^{32}$  bytes. This also may move the upper short section to another memory area. Refer to the loader CHIP command for more information.

### **Long Section**

A section which is not short and which can be placed anywhere in memory.

### **Section Alignment**

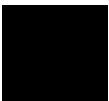
The section alignment attribute may be either 1, 2 or 4. The section alignment attribute affects the beginning address of each file's contribution to a section (i.e., a subsection). That is, if several files each define a relocatable section A, then the beginning address of each section A in each file will be rounded up to a modulo 2 or a modulo 4 boundary if necessary.

### **Section Contents**

There are four section contents indicators:

- Program code (C).
- Data (D).
- Mixed code & data (M).
- ROMable data (R).

The section contents attribute is used by certain HP debuggers in its operation.





## HP Section Type

The HP section type is used to produce HP 64000 symbolic information in the "asmb\_sym" (assembler symbol) and "link\_sym" (linker symbol) files. The HP 64000 file formats define three relocatable sections, PROG, DATA, and COMN as well as the absolute section(s) ABS. The section type attribute is used to map the various relocatable and absolute sections onto the HP 64000 sections PROG, DATA, COMN, and ABS.

---

## Memory Space Assignment

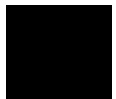
Sections are assigned space in memory in an order which you can control. Also, the initial address (load address) of any or all sections may be specified; this does not alter the order in which sections are assigned space, but it affects the location in memory of following sections which do not have load addresses specified.

Several different kinds of addresses will be referred to in this manual. A *load address* is the memory address at which the lowest byte of a section is placed. A *base address* is the lowest address considered for loading relocatable sections of the absolute object module. Loading need not begin at the base address if SECT and/or COMMON commands are used. A *starting address* is the location at which execution begins. The algorithm used to allocate memory is a three-step procedure as follows:

- 1) Allocate absolute sections and sections specified by the SECT and COMMON linker commands.
- 1) Allocate short sections (= Group I)
- 2) Allocate long sections (= Group II)

The order in which sections are assigned memory within their group is as follows:

- 1 Any sections named in the last ORDER command (for Group II) or SORDER command (for Group I), in the sequence in which they were named in that command.



Linker/Loader Introduction  
**Memory Space Assignment**

- 2 Any other sections belonging to the group, in the sequence in which their names were encountered by the loader.

The loader encounters a name when it appears in a user command or when a module is loaded (with the LOAD command) which refers to that name. Names appear in relocatable object modules produced by the assembler in the sequence in which they appeared in directives in the assembler source input.

Library relocatable object modules which are not selected for inclusion in the absolute object module do not have their section names examined by the loader.

To assign memory to a section, it is necessary to assign it a load address. For those sections whose load addresses you have specified (in a SECT or COMMON directive) nothing more need be done. Otherwise:

- 3 The first short section is loaded at the base address, as specified by the BASE command. If no BASE command is given, the default base address is 0.
- 4 Subsequent short sections are loaded immediately above the preceding section, unless this would cause the high end of the section to extend above \$7FFF, in which case the section is loaded at the lowest address in the short-addressable area of high memory (which depends on the target chip). The loader will not split a short section between low and high memory.
- 5 The first long section is loaded immediately above the short section most recently loaded into low memory. Caution is required because an earlier short section might have been loaded into memory above the most recently loaded short section (if a SECT or COMMON command was used) which will now overlap the long section.

If there are no short sections, the first long section is loaded at the base address specified by the BASE command. If no BASE command is given, the default base address is 0.

- 6 Subsequent long sections start immediately above the preceding long section.

At present, the loader does not support function codes.

## Incremental Linking

The incremental linking feature lets the linker produce a single relocatable object module from several relocatable object modules, resolving all external references between the modules loaded. Undefined external references to other modules can still exist in the output object module. These are reported on the link map.

---

## Relocation Types

By default, sections are word relocatable. That is, they must begin on an even location. (This is true even if an odd load address is specified; in this case the address you supplied will be rounded up.) You may override the default by specifying longword alignment in the SECTION directive.

Also, you may specify via the PAGE and CPAGE commands that certain sections are page relocatable, meaning that their starting address is rounded up to be a multiple of 100. Furthermore, this page relocatability can be turned on and off between modules, which in effect allows you to control the relocation type of each subsection.

Page relocation is useful for debugging since it means the absolute addresses assigned by the loader will match the last two digits of the relocatable addresses shown on the assembler listing.

In the typical load sequence, the loader places contiguously in memory all subsections of the first section it assigns. This is followed immediately by all subsections of the second section, etc. There are no extra bytes between the subsections (unless a subsection contains an odd number of bytes, in which case one byte is left in between the subsections in order that the next higher subsection will start on an even address.)

If any of the subsections specify page relocation, however, the loader will start that subsection at a page boundary to preserve relocation. Due to the internal design of the loader, whenever any subsection is page relocatable, the first subsection also starts on a page boundary, unless a load address is specified for the section. (If paging is in effect at the time the first subsection of a section is LOADED, even a specified load address will be rounded up.)

## Generating HP Format Absolute Files

Since all subsections of a common section start at the same location, specifying page relocation for any common subsection results in page relocation for the section.

---

## Generating HP Format Absolute Files

The assembler provides a command line option to specify that an HP format assembler symbol file be produced for debugging purposes. The linker/loader provides a command line option to specify that HP format absolute and linker symbol files be created.

Problems can arise when generating HP format files. For example, as68k allows periods (.), question marks (?), and dollar signs (\$) in symbol names which are not legal characters in HP format symbols; these characters are converted to underscores (\_) when generating HP format files. Also, the as68k assembler allows symbols up to 31 characters in length while the maximum length of symbols in HP format files is 15 characters; symbols longer than 31 characters are truncated to 15 characters when generating HP format files.

Another problem that can occur when generating HP format files involves the mapping of the large number of sections allowed with this assembler and linker to the three sections (PROG, DATA, and COMN) allowed in HP format files. ld68k uses the HP section type attribute to map relocatable sections as shown below:

```
P ---> PROG
D ---> DATA
C ---> COMN
A ---> ABS
```

Whenever more than one section is mapped to an HP format section, the local symbols in the sections after the first are lost, and a warning is issued. Also, global symbols in sections after the first will become HP format absolute symbols.

## **Return Codes**

ld68k returns 0 if no errors are detected; otherwise, it returns nonzero. The loader will complete normally, issue an informative message, issue a warning, or end abnormally with an error message. Error messages and warnings are listed in the "Loader Error Messages" appendix.

---

## **Loader Listing Description**

The loader uses a two pass process in which the commands and object modules are checked for errors, and a symbol table is formed after encountering the END command. Many errors are not fatal and the loader command processing will continue. The loader will report the errors it encounters with a message immediately following the line in error, and the load will end with the message "LOAD COMPLETED."

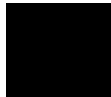
During pass two of processing, the final absolute object module is produced, along with a load map and a listing of unresolved external references. A local symbol table, public symbol table, and cross reference table may be listed in the load map. The load map also indicates the starting address of the load, as well as the output module name and format, and the section and module summary. Detailed descriptions of the map file are found in the following "Loader Listings" section.

---

## **Loader Listings**

Note the following points when examining loader listings.

- 1 The first page of the listing shows all commands which you have entered along with any command errors that occur.
- 2 The next page begins the load map which first displays the output module name and the output module format. The load map next displays the names of all sections followed by the attribute, starting address, ending address, length and type of alignment for each section. Then, the load map



displays a module summary containing the names of all the modules followed by the starting address and ending address for each section in each module. Any executable address errors encountered during pass 2 of the load are indicated at the end of the module summary.

- 3 When the appropriate LIST command options are specified, lists of all local symbols and public symbols are displayed in symbol tables. All symbols in the map are truncated to 10 characters. Public symbols are external definition symbols as declared in the assembler, and are used for intermodule communication. Local symbols are those known to only a single module. Local symbols are not used by the loader, but are listed so their final absolute values may be seen. The attributes and sections are listed for each local symbol, as well as the section offsets and modules which define them. If the cross reference list option is specified, a cross reference table is listed. Local symbols may be placed in the output object module of the assembler by specifying the "LIST S" directive, and may subsequently be used for symbolic debugging.

- 4 The local symbol table contains two types of symbols:

High level elements are compiler symbols whose attribute is LOCAL. The OFFSET column indicates the stack address offset in bytes for each section. High level symbols contain both MODULE and FUNCTION information.

Low level elements are assembler symbols whose attribute is ASMVAR. OFFSET is the actual section address. Only the MODULE information is listed in the local symbol table.

- 5 The public symbol table contains the list of PUBLIC symbols, the section, the actual section address, and the modules.
- 6 The unresolved externals section contains a list of the undefined external references.
- 7 The cross reference option is turned off by default. To produce the cross reference table, use the "LIST C" command. All external symbols passed to the loader are listed under the heading "SYMBOL". The symbol section and address are listed. Any flag to the left of those values is the segment attribute of the symbol.

Under "MODULE", a module name preceded by a minus sign indicates that the symbol was defined in that module. Line numbers not preceded by a minus sign indicate a reference to the symbol in that module.

- 8 Next, the starting address of the load is indicated.

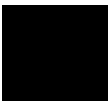
- 9 Finally, the end of the load is indicated by the "LOAD COMPLETED" or "LOAD NOT COMPLETED" message.

Sometimes the module listed for a public symbol will be \$\$\$. \$\$\$ indicates that the symbol does not belong to any module. \$\$\$ symbols occur in the following situations:

- 10 Linker defined symbols. The PUBLIC, INDEX, and INITDATA command cause the linker to define symbols.
- 11 Undefined symbols.
- 12 Common section names.
- 13 Global symbols whose value is outside of any section. Usually this is a result of EQUing a symbol to a constant value.



Linker/Loader Introduction  
**Loader Listings**





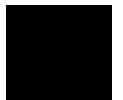
---

# 10

---

## Linker/Loader Commands

This chapter describes each of the linker/loader commands.



## Linker/Loader Commands

The loader reads a sequence of commands from a linker command file or from standard input. The last command must be either an EXIT or END command.

The object modules are read from files specified in the LOAD command.

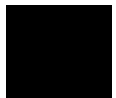
The output of the loader consists of an absolute load module suitable for loading into an actual microprocessor. The output module is written in one of several absolute object module formats as described in this chapter's FORMAT description.

### Summary of Commands

The following pages describe the loader commands. In the command descriptions, brackets, { }, are used to indicate optional arguments. A summary of the commands is given below:

;	(Comment)	Specify Comment.
#	(Continuation)	Line continuation character.
ABSOLUTE		Specify the Sections Included in the Absolute File.
ALIAS		Specify Section Assumed Name.
ALIGN	{MOD}	Sets Alignment for Named Section.
BASE		Specify Location at which to begin Loading.
{LOWER}	CASE	Shifts Names to Lower Case.
{UPPER}	CASE	Shifts Names to Upper Case.
CHIP		Specify Target Microprocessor.
COMMON		Set Common Section Load Address.
CPAGE		Set Paging for Common Section.
{NO}		Retains or Discards Internal Symbols.
DEBUG	_SYMBOLS	
END		End Command Stream and Finish Loading.

ERROR	Change Message Severity to ERROR.
EXIT	Exit Loader.
EXTERN	Creates External References.
FORMAT	Specify Absolute File Format.
INCLUDE	Includes a Command File.
INDEX	Give Loader the Run-Time Value of Register "An".
INITDATA	Specify ROM Address for Section.
{NO}INTFILE	Stores Information Using Intermediate File or Virtual Memory.
LIST	Set Loader Options.
LISTABS	Lists Symbols to Output Object Module.
LISTMAP	Specifies Layout and Content of the Map.
LOAD	Load Specified Object Modules.
LOAD_SYMBOLS	Load Object Module Symbol Information.
MERGE	Combines Named Module Sections.
NAME	Specify Output Module Name.
NLIST	Clear Loader Options.
NOERROR	Change Message Severity to NOERROR.
NOPAGE	Turn off Paging for Section.
ORDER	Specify Long Section Order.
PAGE	Set Paging for Noncommon Section.



## Linker/Loader Commands

PUBLIC	Specify PUBLIC symbols (External Definitions).
RESADD	Reserves Region of Memory.
RESMEM	Reserves Region of Memory
SECT	Set Noncommon Section Load Address.
SECTSIZE	Set Minimum Section Size.
SORDER	Specify Short Section Order.
START	Specify Output Module Starting Address.
WARN	Change Message Severity to WARNING.

## Command Format

Commands may begin in any column. Command arguments may follow in any column and must be separated from the command by at least one blank. Comments may follow commands as long as a semicolon separates the command from the comment. Entire lines in the command stream may be commented with a semicolon as the first nonblank character in the line.

Numeric command arguments may be either decimal or hexadecimal and may be represented in either of the following two ways:

- 1 Hexadecimal constants may be preceded by a "\$" (e.g., \$1F) in which case they need not have a leading zero even if they start with hexadecimal characters A - F. Any legal hexadecimal constant may be used, and a terminator is not required.
- 2 Hexadecimal constants may be terminated by the letter "H", or the letter "X" (e.g., 1FX), in which case any legal hexadecimal constant may be used in the command argument, and a leading \$ is illegal.

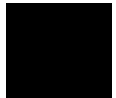
Symbols and section names must follow the syntax rules for symbols given in the assembler manual: i.e., they must begin with a letter, a period ".", a question mark "?", or an underbar "\_", and subsequent characters may be any of these, a dollar sign "\$", or a decimal digit.

Section names, symbols, and module names are case sensitive by default. The LOWERCASE or UPPERCASE commands may be used to alter this. The assembler directive OPT CASE may be used to specify that symbols are case sensitive in the assembly.

## Processing Order

The linker will process commands in the following order and also handle positional dependencies by the following rules:

- 3 Preprocessed commands such as:
  - INCLUDE
 are expanded before any linker commands are processed.
- 4 {NO}INTFILE must be before any LOAD command.
- 5 Non-position dependent commands are processed next.
  - BASE
  - CHIP



## Linker/Loader Commands

FORMAT  
NAME  
START  
LISTABS  
LISTMAP  
RESADD  
RESMEM

- 6 Position-dependent commands are processed next.

CASE

should be before any command using names

LOWERCASE

should be before any command using names

UPPERCASE

should be before any command using names

{NO}PAGE

CPAGE

{NO}DEBUG\_SYMBOLS

LOAD

EXTERN

- 7 Commands that are position-independent in the command file are processed next, but they are operated on in the following order:

- a. COMMON, SECT, PUBLIC
- b. MERGE
- c. ALIAS
- d. ORDER, SORDER
- e. ABSOLUTE, INDEX, INITDATA

- 8 Commands that end command processing are processed last.

END  
EXIT

## ; (Comment)

### Specify Loader Comment

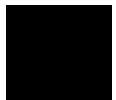
A semicolon may be used to specify a comment in the command stream.

Comments may be used to document loader command sequences. Comments following loader commands must be separated from the command with a semicolon. Entire lines may be commented by using the semicolon as the first nonblank character in a line.

Note that the comment character has changed from the \* (asterisk) used by some previous HP loaders.

### Example:

```
; LOADER COMMENT EXAMPLE  
BASE $1000 ; Another comment.
```



## **# (Continuation)**

### **Continue Command**

The # (pound sign) character may be used to continue a command from one line to the next. This is particularly useful in ORDER commands containing a large number of module names. The linker treats all characters and the end-of-line following the # character as a single blank followed by the first character of the next line.



---

## ABSOLUTE

### Specify the Sections Included in the Absolute File

**Syntax:**

Command	Argument
ABSOLUTE	sname{,sname} . . .

**Where:**

sname            Is the name of a relocatable section to be put into the output file.

**Description:**

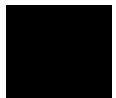
The ABSOLUTE command allows you to specify that only the code and data from certain, specified program sections be included in the output file. Without the ABSOLUTE command, all code and data from all sections in all the input modules is put into the absolute output file.

The ABSOLUTE command allows you to use of code overlays. Typically, in an application employing overlays, there is a main code section and several "overlay" sections. Usually, the main section stays in memory. The overlays are not resident but are loaded into memory as needed during the execution of the program. However, the overlay sections need to be linked with the main section.

When using the ABSOLUTE command, only code and data from relocatable sections is ever put into the output. Code and data from absolute (i.e., ORGed) sections is never put into the output when the ABSOLUTE command is used.

**Example:**

The following example shows how to link an application containing overlays. It requires three link operations and three linker command files.



## Linker/Loader Commands

### ABSOLUTE

The program consists of a main program and two overlays. All the code and data for the main section is in section "MAINSECT". All the code for first overlay is in section "OV1SECT" and all the code for the second overlay is in section "OV2SECT".

Linker command file for main section:

```
SECT    MAINSECT=$1000      ; Locate the main section.
SECT    OV1SECT=$2000      ; Locate first overlay.
SECT    OV2SECT=$2000      ; Second overlay will cause ERROR: Section Overlap.
ABSOLUTE MAINSECT          ; Only this section goes into output file.
LOAD    MOD1,MOD2,...,MODn ; Load all modules for main, overlay 1, overlay 2.
END
```

Linker command file for first overlay section:

```
SECT    MAINSECT=$1000      ; Locate the main section.
SECT    OV1SECT=$2000      ; Locate first overlay.
SECT    OV2SECT=$2000      ; Second overlay will cause ERROR: Section Overlap.
ABSOLUTE OV1SECT          ; Only this section goes into output file.
LOAD    MOD1,MOD2,...,MODn ; Load all modules for main, overlay 1, overlay 2.
END
```

Linker command file for second overlay section:

```
SECT    MAINSECT=$1000      ; Locate the main section.
SECT    OV1SECT=$2000      ; Locate first overlay.
SECT    OV2SECT=$2000      ; Second overlay will cause ERROR: Section Overlap.
ABSOLUTE OV2SECT          ; Only this section goes into output file.
LOAD    MOD1,MOD2,...,MODn ; Load all modules for main, overlay 1, overlay 2.
END
```

## ALIAS

### Specify Section Assumed Name

**Syntax:**

Command	Argument
ALIAS	sname,alias_sname

**Where:**

sname	Specifies the section name.
alias_sname	Specifies the name of the section which is to be considered the same as "sname".

**Description:**

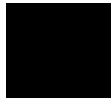
The ALIAS command allows you to specify that a certain section be considered the same as another section. The alias command is useful in that you can cause the loader to load the parts of those sections contiguously, as if they were parts of the same section. The resulting output object file will show the two combined sections under the alias name. Without the ALIAS command, the loader would load the parts of those two sections in separate areas.

The ALIAS command is similar to the MERGE command in that it can combine differently named sections. However, the ALIAS command can only combine two sections, and does so in the order they appear in LOAD commands. The MERGE command can combine more than two sections and combines them in the order specified in the MERGE command. MERGE and ALIAS are mutually exclusive and cannot appear in the same link session.

If this command is used, it must be specified before any LOAD commands.

**Example:**

```
ALIAS      SECT1 , SECT2
```



## **ALIGN{MOD}**

### **Sets Alignment for Named Section**

**Syntax:**

Command	Argument
ALIGN{MOD}	sname= align_value

**Where:**

sname	A section name.
align_value	A constant which is a power of 2 between 1 and $2^{32}$ .

**Description:**

Every relocatable module section has an alignment attribute. When the module section is located, its base address is made a multiple of the alignment by the linker.

The ALIGNMOD command may be used to increase the alignment attribute of the module sections of the named module. Note that the alignment of a given combined section is the largest of its inclusive module sections.

The ALIGN command sets the alignment of the beginning of the combined section only. If any of the module subsections that make up the combined section has an alignment that exceeds the setting, a warning will be generated and the combined section will have the greater alignment.

---

## **BASE**

### **Specify Location at Which to Begin Loading**

**Syntax:**

Command	Argument
BASE	number

**Where:**

number            An absolute number.

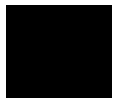
**Description:**

The BASE command specifies the lowest address where the loader will place a relocatable section, provided the section does not have its load address specified in a SECTION or COMMON command. You may find the BASE command useful to avoid collision with an operating system, for example, in low memory.

The BASE address must be an absolute number.

**Example:**

```
BASE            $400
```



## [UPPER]CASE, [LOWER]CASE

### Controls Case-Sensitivity

#### Syntax:

Command	Argument
CASE	{class{,class}...}
LOWERCASE	{class{,class}...}
UPPERCASE	{class{,class}...}

#### Where:

class            One of the following:

                  PUBLICS  
                  MODULES  
                  SECTIONS

#### Description:

The CASE command controls the case-sensitivity of various classes of symbolic names.

Each of the functions of the CASE command are described below:

- CASE without the prefix UPPER or LOWER specifies that upper and lower-case characters are distinct in name comparisons. Symbolic names in the indicated class(es) are not modified on input.
- LOWERCASE causes the linker to shift names to lower case on input. All symbolic names of the specified class(es) will appear in lower case in the linker's output files.
- UPPERCASE causes the linker to shift names to upper case on input. All symbolic names of the specified class(es) will appear in upper case in the linker's output files.

The CASE, UPPERCASE, or LOWERCASE commands affect only the classes of names specified by the class option. If class is not specified, all classes of names are affected. Each class can have only one case specification (i.e. CASE, UPPERCASE, or LOWERCASE).

## Linker/Loader Commands [UPPER]CASE, [LOWER]CASE

The PUBLICS class refers to all the public and external names. The SECTIONS class refers to all the section names, and the MODULES class refers to all the module names.

CASE, LOWERCASE, and UPPERCASE take immediate effect and should be early in the command file.

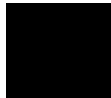
### Example:

Given the following command file:

```
UPPERCASE PUBLICS
LISTMAP   PUBLICS
LOAD      modulea, moduleb, modulec
END
```

All public and external names will be upper-cased in the linker's output file. The generated link map will contain a PUBLIC SYMBOL TABLE section that will show all the upper-case public and external names. For example:

```
. . .
PUBLIC SYMBOL TABLE
SYMBOL          SECTION          ADDRESS          MODULE
G1               sect3             00001200         MODULEA
G2               sect3             00001204         MODULEB
G3               sect3             00001208         MODULEC
. . .
```



## **CHIP**

### **Specify Target Microprocessor**

**Syntax:**

Command	Argument
CHIP	target {,n}

**Where:**

target	An expression evaluating to 68000, 68EC000, 68HC000, 68HC001, 68008, 68010, 68302, 68330, 68331, 68332, 68333, 68340, CPU32, 68020, 68EC020, 68030, 68EC030, 68040, or 68EC040.
n	The bus width parameter.

**Description:**

The CHIP command declares the microprocessor on which the linked code is to run. The CHIP command may specify the 68000, 68EC000, 68HC000, 68HC001, 68008, 68010, 68302, 68330, 68331, 68332, 68333, 68340, CPU32, 68020, 68EC020, 68030, 68EC030, 68040, or 68EC040. The differences are the instructions allowed, the size of the address space, and the addresses of the high memory area which can be accessed with Absolute Short address mode. The linker places sections with the Short attribute only in this area of memory (or in the low short-addressable area of memory, which is from 0 to \$7FFF for all targets). If no CHIP command appears, the target microprocessor is taken to be the one from the input modules with the greatest capability. For example, if three modules specify 68000, 68010, and 68020 respectively, the default will be taken to be 68020.

The 68010 implements more instructions than the 68000 or 68008 (which have the same instruction set). The 68020 implements more instructions than the 68010. The 68030 and 68040 implement additional instructions over the 68020. In order to prevent an illegal opcode, the loader issues an error if a module is loaded whose CHIP has greater capabilities than the CHIP specified to the loader.

The differences between the various chips are summarized below:



CHIP	Maximum Address	High short-addressable area of memory	
-----	-----	-----	-----
68000	\$FFFFFF	\$FF8000	to \$FFFFFF
68CH001	\$FFFFFF	\$FF8000	to \$FFFFFF
68010	\$FFFFFF	\$FF8000	to \$FFFFFF
68302	\$FFFFFF	\$FF8000	to \$FFFFFF
68331	\$FFFFFF	\$FF8000	to \$FFFFFF
68332	\$FFFFFF	\$FF8000	to \$FFFFFF
68008	\$FFFFF	\$F8000	to \$FFFFF
68020	\$FFFFFFFF	\$FFFF8000	to \$FFFFFFFF
68030	\$FFFFFFFF	\$FFFF8000	to \$FFFFFFFF
68040	\$FFFFFFFF	\$FFFF8000	to \$FFFFFFFF
CPU32	\$FFFFFFFF	\$FFFF8000	to \$FFFFFFFF

If present, the CHIP command must precede all other loader commands.

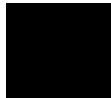
The CHIP command may also specify an optional bus width parameter to override the maximum bus width implied for the target microprocessor. This allows the maximum address in memory to be limited regardless of the bus width possible for the chip. Limiting the bus width may also change the location of the high short section of memory.

All absolute addresses which appear in later commands or object modules are checked against the bounds established by the CHIP command.

The bus width parameter allows you to specify a maximum address up to  $2^{n-1}$  and a high short-addressable area address range from  $2^n - \$8000$  to  $2^n - 1$ .

**Example:**

```
CHIP      68020,24
```



## **COMMON**

### **Set Common Section Load Address**

**Syntax:**

Command	Argument
COMMON	sname,value
COMMON	sname= value
COMMON	sname value

**Where:**

sname	Specifies the section name.
value	Specifies the load address of the common section.

**Description:**

This command is used to specify the load address of a common section. If this command is used it must be specified before any LOAD commands.

If this is the first occurrence of this section name it is given the attributes common and long.

Specify the section name followed by the address at which to begin loading the section. The address specified is always rounded up to the next higher word boundary, and to the next higher page boundary if paging is specified for this common section.

**Example:**

```
COMMON    COMSEC , 2048
```

---

**Note**

The value is separated from the section name by a blank, comma, or equal sign. Multiple COMMON commands with the same section name are accepted without a warning, but only the last one will be used.

---

---

## **CPAGE**

### **Set Common Section to be Page Relocatable**

**Syntax:**

Command	Argument
CPAGE	sname

**Where:**

sname            A section name.

**Description:**

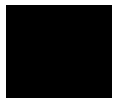
This command may be used to modify the relocation type of common section(s) in the input object modules to Page. It allows you to override the default relocation type of Word for a common section.

Since all subsections of a common section are loaded at the same address, the CPAGE command need only be used once per section at the beginning of the loading process. Once Page Relocation is turned on for a common section, it cannot be turned off for later subsections of the section.

Specify the section name. If this is the first occurrence of the section name, it is assigned the attributes common and long.

**Example:**

```
CPAGE            P
```



## **[NO]DEBUG\_SYMBOLS**

### **Retains or Discards Internal Symbols**

**Syntax:**

Command	Argument
{NO }	DEBUG_SYMBOLS

---

**Description:**

These commands control putting local symbols into Motorola S-Record output files. These commands may be placed between LOAD commands to selectively copy symbols from certain modules. DEBUG\_SYMBOLS is a synonym for the LIST P command and NODEBUG\_SYMBOLS is a synonym for the NLIST P command.

## **END**

### **End Command Stream and Finish Loader**

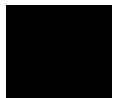
**Syntax:**

Command	Argument
END	

---

**Description:**

This command should be the last command in every command stream. It initiates the final steps in the load process. END completes the load, produces an output object module, and returns to the operating system.



## **ERROR, WARN, NOERROR**

### **Modify Message Severity**

**Syntax:**

Command	Argument
ERROR	condition{condition} ...
WARN	condition{condition} ...
NOERROR	condition{condition} ...

**Where:**

condition      One of UNREF, UNRES, OVERLAP, DUPLIBPUB, or a number corresponding to the message number of the error or warning.

**Description:**

These commands change the way a message or group of messages is treated. ERROR causes the message to be treated as an error; WARN causes the message to be treated as a warning; NOERROR causes the message to be treated as a non-error (that is, the message is ignored).

The ERROR, WARN, and NOERROR commands affect all messages which are generated after the linker encounters the command. The change in message severity remains in effect until the linker has finished processing. The effect of these commands cannot be changed by subsequent ERROR, WARN, or NOERROR commands.

Fatal errors and messages generated by the ERROR, WARN, or NOERROR command cannot be overridden or modified.

## **EXIT**

### **Exit Loader**

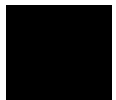
**Syntax:**

Command	Argument
EXIT	

---

**Description:**

The EXIT command is like the END command in that it is the final command in the linker command file. The EXIT command differs from the END command in that it prevents the final output from occurring. All object modules are read and all linker commands are processed and checked for errors, but no output module is generated.



## **EXTERN**

### **Creates External References**

#### **Syntax:**

Command	Argument
EXTERN	name{,name}...

#### **Where:**

name                    The symbolic name of an external reference that is to be created.

#### **Description:**

The EXTERN command creates external references for the linker to resolve. The EXTERN command can appear anywhere in a command file.

Multiple EXTERN commands can appear in a command file.

The EXTERN command is in effect for a given name when that name is specified in the command. It remains in effect until the end of the command file, but it has no effect before the point of specification. An EXTERN command with a specific name must appear before the LOAD command for the library in which the specific external symbol is defined in order to force the loading of the module associated with the external symbol.

The -u name command line option has the same effect as if an EXTERN command is inserted into the command file before the first LOAD command, if any.

#### **Example:**

```
EXTERN g1
LOAD module1.o,module2.o,extern.lib
END
```

In this example, the symbol g1 is not referenced in either module1 and module2. So if a definition of g1 exists in extern.lib, the library module that contains the definition will be loaded to resolve the external reference.



---

## **FORMAT**

### **Specify Absolute File Format**

**Syntax:**

Command	Argument
FORMAT	option

**Where:**

option is one of the following:

S	Motorola S absolute hexadecimal.
IEEE	HP-MRI IEEE-695 absolute output format.
INCREMENTAL	IEEE relocatable format
HP	HP 64000 absolute and linker symbol file format.
NOABS	No output file. This is the same as the NLIST O command.

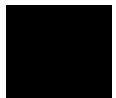
**Description:**

The FORMAT command may be used to specify which output absolute object module format the loader is to produce. Option switches may be set to produce absolute IEEE (default), Motorola S-record, HP 64000 format, IEEE relocatable, or no output file at all.

**Example:**

```
FORMAT      S
```

In this example, the loader produces an absolute load in Motorola S-Record output format.



## **INCLUDE**

### **Includes a Command File**

#### **Syntax:**

Command	Argument
INCLUDE	filename

#### **Where:**

filename            The file to be included in the linker command file.

#### **Description:**

The INCLUDE command lets additional command files be included in a linker command file. At the point the INCLUDE command is specified, the text contained in the file specified by filename is included in the linker command file.

The INCLUDE command can appear multiple times anywhere in a linker command file and can be nested up to a maximum depth of 16.

#### **Example:**

If setup.opt contains:

```
CHIP 68010
BASE $500
```

and a command file contains the following INCLUDE command:

```
INCLUDE setup.opt
LOAD module1,module2
```

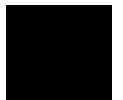
the resulting link map will be:

```
. . .
. . .
INCLUDE setup.opt
CHIP 68010
BASE $500
*** End of include file: /some/where/setup.opt
LOAD module1,module2
. . .
. . .
```

An extra comment line:

```
*** End of include file: /some/where/setup.opt
```

with the absolute path name of the included file was added by the linker for readability.



## **INDEX**

### **Specify the Run-Time Value of Register "An"**

**Syntax:**

Command	Argument
INDEX	?REGn, SECTNAME, OFFSET

**Where:**

REGn	Is one of the address registers: A2, A3, A4, or A5.
SECTNAME	The name of a relocatable section whose load address (plus an optional offset) is specified to equal the run-time value of address register REGn.
OFFSET	A number to be added to the load address of the relocatable section specified. The result is specified to be the run-time value of REGn.

**Description:**

The INDEX command is used to inform the loader of the run-time value of an address register "An" (where n = 2, 3, 4, or 5). The value you associate with a particular "An" register will equal a relocatable section's load address plus an offset value.

---

**Note**

---

The only A registers which may use the INDEX command are A2, A3, A4, or A5.

A public symbol, equal to the run-time value specified, will be created in the form "?An". This symbol can be declared as an external symbol in the assembly language source file (with the XREF directive) and used to initialize the appropriate address register.

## Purpose of the INDEX Command

The loader needs to know the run-time value of an address register whenever you use assembly language operands which combine relocatable expressions and address register indirection. For example, consider the following assembler syntax:

< rel exp> (An) or (< rel exp> ,An)

Operands of the form shown above will generate the **Address Register Indirect with Displacement** address mode which requires a 16-bit displacement. The relocatable expression in the syntax above is an effective address or, in other words, the location to be accessed. The loader must calculate the 16-bit displacement using the equation:

$$\begin{aligned} \text{< ea>} &= \text{An} + \text{disp} \\ \text{disp} &= \text{< ea>} - \text{An} \\ \text{disp} &= \text{< rel exp>} - \text{An} \end{aligned}$$

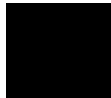
The INDEX command makes "An" a known value which enables the loader to calculate the displacement. If the INDEX command is not used, the loader will calculate the displacement under the assumption that the run-time value of the address register is zero.

Other address modes which can contain relocatable expressions in conjunction with address register indirection are the 68020 model modes: **Address Register Indirect with Base Displacement and Index**, **Memory Indirect Post-Indexed**, and **Memory Indirect Pre-Indexed**.

### Example:

```
INDEX    ?A2,DATA1,8000H    * This offset allows "(A2)" indirect
                                * addressing to access a full 64K bytes
                                * in section DATA1 (using a 16-bit
                                * signed displacement).
```

See the "A2-A5 Relative Addressing" chapter for additional information on how the INDEX command may be used with array addressing for registers A2 through A5.



## INITDATA

### Specify Initialized Data in ROM

**Syntax:**

Command	Argument
INITDATA	merge_arg [,merge_arg] ...

**Where:**

merge\_arg      May be any of the following:

sectname  
or {sectname,module}  
or {\*,module}

The first form copies data from section *sectname*. The second form copies data from the portion of *sectname* defined in *module*. The third form copies data from all sections defined in *module*.

**Description:**

The INITDATA command provides a method to copy data from ROM into RAM before a program is executed.

INITDATA causes the linker to create a new data section called ??INITDATA. The data from the sections named in the command string is copied into the ??INITDATA section.

The user program must call the `initcopy()` routine at run time to reinitialize the data in RAM each time the program runs. The `initcopy()` routine checks the special bytes generated by the linker in the section ??INITDATA to provide the necessary information: copy destination address, copy size, and data.

The ??INITDATA section may be ordered and assigned an address using standard commands.

### Example:

The following example will cause the linker to create the section `??INITDATA` at link time which contains all of the section contents for `sec1`, `sec3` and `sec4` so that they will be copied to a specified address at run time:

```
INITDATA sec1,sec3,sec4
```

The section name may be qualified by a module name and type, as in

```
INITDATA sec2,{module2,DATA}
```

More examples of the `INITDATA` command are supplied in directory

```
/usr/hp64000/demo/languages/B3641/  
features/INITDATA
```

on UNIX systems, or in the examples directory on DOS systems.

### Initcopy

The `initcopy` routine is supplied with the assembler in the library file `/usr/hp64000/lib/68000/initcopy.s` (`\hpas68k\initcopy.s` on DOS systems). If you have special needs, it is possible to write your own `initcopy` routine.

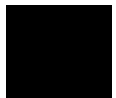
#### Supplied `initcopy` routine

The `initcopy` routine has been supplied in two forms: C source (`.c`), assembly code from C (`.s`), and object code (`.a`). The object code form is supplied as a library in the file `/usr/hp64000/lib/68000/as68xxx.a` (UNIX) or `\hpas68k\as68xxx.lib` (DOS). On UNIX systems, the source code forms are in the `src` subdirectory.

`Initcopy` calls the C function `memcpy()`. The assembly source for `memcpy` is supplied in the file `memcpy.s`.

#### Writing your own `initcopy` routine

The virtual address of the ROM section can be set using the `.STARTOF` operator.



Linker/Loader Commands  
**INITDATA**

The data in the ??INITDATA ROM section uses the following special bytes. These bytes are generated by the linker, and may be used by your initcopy routine.

**S** Start of operation. It should be immediately followed by one of the other special bytes.

**C** Copy. After this byte, you need to include the total number of bytes which need to be copied, the destination address for the data, then the data itself:

	4 bytes	4 bytes	<i>length</i> bytes
<b>C</b>	<i>length</i>	<i>destination</i>	<i>data</i>

**E** End of operation.

**R** Repeat pattern. Not currently implemented. After this byte, you need to include the repeat count, the destination address, the size of the pattern, and then the pattern:

	4 bytes	4 bytes	2 or 4 bytes	<i>size</i> bytes
<b>R</b>	<i>count</i>	<i>destination</i>	<i>size</i>	<i>pattern</i>

**B** Byte repeat. Not currently implemented. After this byte, you need to include the repeat count, the destination address, and then the byte to be repeated:

	4 bytes	4 bytes	
<b>B</b>	<i>count</i>	<i>destination</i>	<i>byte</i>

The R and B bytes are not implemented in the linker at this time. They may be implemented in a later version of the linker to provide data compression.



---

## **[NO]INTFILE**

**Stores Information Using Intermediate File or Virtual Memory**

**Syntax:**

Command	Argument
{NO }	INTFILE

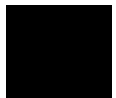
**Description:**

The linker, like the assembler, is a two pass program. Intermediate information is stored, by default, using virtual memory between pass 1 and 2. The INTFILE command lets you store this intermediate information in a temporary file. The NOINTFILE command lets you store this information using virtual memory.

With different systems, using a temporary file may be faster than using virtual memory. Also, depending on the configuration for running large jobs, the virtual allocation size can be limited. You can try to run the program using the INTFILE command which then produces an intermediate file as opposed to using virtual memory.

**Example:**

```
INTFILE
LOAD mod1.obj
LOAD mod2.obj
END
```



## **LIST**

### **Set Loader Options**

**Syntax:**

Command	Argument
LIST	FLAG {,FLAG} . . .

**Where:**

flag is one of the following:

- A Specifies the output file format to be Motorola S-Record. Same as FORMAT S. (Default= IEEE.)
- C Specifies that a cross reference listing is to be produced. Same as LISTMAP CROSSREF. (Default= NLIST C, i.e., no cross reference)
- H Specifies HP 64000 format absolute and linker symbol output files. Same as FORMAT HP. (Default = IEEE.)
- I Specifies the output file format to be IEEE. Same as FORMAT IEEE. (Default= IEEE.)
- O Specifies that an object module is to be produced. NLIST is the same as FORMAT NOABS. (Default= LIST O)
- P The P flag only affects Motorola S-Record output files and is therefore only effective when S-records are selected and the S flag is in effect. The P flag specifies that the local symbols from input modules loaded (while this flag is set) be included in the output file. This flag can be turned off and on between LOAD commands. Its purpose is to exclude local symbols from particular modules because of duplicate symbol conflicts. Same as DEBUG\_SYMBOLS. (Default= LIST P)
- S Specifies that the local symbol table information be written to the output file for debugging. The effect of the S flag

depends upon what output format is selected. Same as LISTABS INTERNALS. (Default= LIST S)

If the output is Motorola S-records, then the S flag causes symbols and their values to be written at the beginning of the S-record file. NOLIST S suppresses the writing of these symbols.

If the output is IEEE-695, the S flag causes local assembly symbols and compiler-generated symbol and type information to be written to the IEEE file. NOLIST S suppresses this information. Global assembly symbols (for instance, those mentioned in XDEF directives) are always written to the IEEE file regardless of any flag.

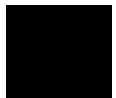
If the output is HP 64000 format, the S flag has no effect on the link\_sym file.

- T Specifies that the local symbol table be listed on the loader listing. Same as LISTMAP INTERNALS. (Default= NLIST T, i.e., off)
- X Specifies that the PUBLIC (global) symbol table be listed on the loader listing. Same as LISTABS PUBLICS. (Default= NLIST X, i.e., off)

**Description:**

The LIST command may be used to change the loader internal flags. These flags control the format and contents of the output file, as well as the contents of the loader listing. The LIST options specified will remain in effect through all modules until another LIST or NLIST command is encountered.

All of the flags have equivalent commands which perform the same function.



Linker/Loader Commands  
**LIST**

**Example:**

```
LIST      T,X      ; list both local and  
           ; definition symbol tables
```

---

**Note**

Though the LIST command is available, it is a better choice to use LISTABS and LISTMAP.

---

## LISTABS

### Lists Symbols to Output Object Module

**Syntax:**

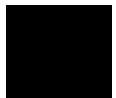
Command	Argument
LISTABS	option{,option}...

**Where:**

option	One of the following:
{NO}PUBLICS	Places globally-defined symbols into the output object module. LISTABS NOPUBLICS prevents globally-defined symbols from being placed in the output object module. (Default: NOPUBLIC)
{NO}INTERNALS	Places the internal (local) symbols in the output object module and omits any symbols that are defined in modules for which the NODEBUG_SYMBOLS command is in effect. LISTABS NOINTERNALS suppresses the placement of internal symbols into the output object module. (Default: INTERNALS)

**Description:**

The LISTABS command controls the output of certain items to the output object module. Multiple LISTABS commands can be specified and have a cumulative effect. Options that are inconsistent with previous LISTABS commands cannot be specified in a succeeding LISTABS command. For example, LISTABS PUBLICS cannot be followed by LISTABS NOPUBLICS, but can be followed by LISTABS INTERNALS.



## **LISTMAP**

### **Specifies Layout and Content of the Map**

**Syntax:**

Command	Argument
LISTMAP	option{,option}...

**Where:**

option is one of the following:

{NO}CROSSREF	Causes a cross-reference listing to be output to the map file. NOCROSSREF suppresses the generation of this cross-reference listing. (Default: NOCROSSREF)
{NO}INTERNALS	Causes a listing of the non-public (local) symbol table to be output to the map file. NONINTERNALS suppresses the output of the non-public symbol table. (Default: NOINTERNALS)
{NO}PUBLICS	Causes a listing of the public symbol table to be output to the map file. NOPUBLICS suppresses the output of the public symbol table. (Default: NOPUBLICS)
LENGTH lval	Sets the map file page length. The range for <i>lval</i> is 5 to 255. (Default: LENGTH 55)

**Description:**

The LISTMAP command controls the output of certain items to the linker's map file. Multiple LISTMAP commands can be specified and have a cumulative effect. Options that are inconsistent with previous LISTMAP commands cannot be specified in a succeeding LISTMAP command.

---

**Note**

LISTMAP CROSSREF was formerly known as LIST C. LISTMAP INTERNALS was formerly known as LIST T, and LISTMAP PUBLICS was formerly known as LIST X.

---

## LOAD

### Load Specified Object Modules

**Syntax:**

Command	Argument
LOAD	{-}filename1 {,-}filename2,...,{-}filenameN

**Where:**

filename            Specifies the name of a file in which the object module or library resides. If the filename contains a suffix, it is used as is; otherwise, a suffix of ".o" or ".obj" (DOS) is appended to form the actual filename. The minus sign in front of the filename forces the linker to load all modules in *filename*.

**Description:**

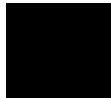
The LOAD command is used to specify one or more input object modules to be loaded.

The file specified may contain either relocatable object modules (output of the assembler), relocatable object modules from incremental linking, or libraries (output of the librarian). Libraries and object modules differ in their internal format, but you can treat them identically with the following exception:

Libraries should be loaded after all non-libraries. Libraries will load only those modules which are necessary to resolve undefined XREFs, even if the library file or device is preceded by a minus sign. Backward XREFs within a library are resolved correctly. However, XREFs to a library from a subsequently LOADED file are generally not resolved correctly.

Therefore, libraries should be LOADED last. In the case where each of two libraries makes XREFs to the other, it is generally necessary to LOAD one of them twice (for example, LOAD LIBA, LIBB, LIBA) to pick up all the necessary modules.

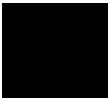
The object modules are loaded in the order specified with each subsection within each module being loaded into memory at a higher address than all preceding subsections within its section. You may use as many LOAD commands as needed.



Linker/Loader Commands  
**LOAD**

**Example:**

```
LOAD FILE1
```





---

## LOAD\_SYMBOLS

### Load Object Modules Symbol Information

#### Syntax:

Command	Argument
LOAD_SYMBOLS	{-}filename1{,-}filename2,...{,-}filenameN}

#### Where:

filename	Specifies the name of a file in which the object module or library resides. If the filename contains a suffix, it is used as is; otherwise, a suffix of ".o" or ".obj" (DOS) is appended to form the actual filename. The minus sign in front of the filename forces the linker to load all modules in <i>filename</i> .
----------	--

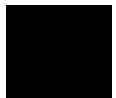
#### Description:

The LOAD\_SYMBOLS command allocates space internally to the linker for modules contained in the specified file(s) for so that it may correctly fix up symbols. The linker also retains all PUBLIC symbol definitions. Code and data for the specified modules are not loaded, but symbol table and debug information is loaded into the output file.

Input modules can consist of relocatable object modules (output of the assembler), relocatable object modules from incremental linking, or libraries (output of the librarian). If the specified modules are from a library, all external symbols are also retained so that all forward references cause an allocation of space. If the modules are not from a library, all external symbols are ignored.

#### Example:

```
LOAD_SYMBOLS    FILE1
```



## MERGE

### Specify Output Module Name

#### Syntax:

Command	Argument
MERGE	merge_name merge_arg{,merge_arg}...

#### Where:

merge\_name      The name of the new, combined section.

merge\_arg is one of the following:

sname            a section name

{sname,mname}   a section name followed by a module name. The braces are required.

\*                an asterisk can replace either or both the section name and the module name. A wild card character, an asterisk means all modules or sections.

#### Description:

The MERGE command renames all the named subsections to a new section named in the first argument. This command lets you overcome the default combining of sections with the same name section by letting you create new sections. The command lets you concatenate arbitrary lists of subsections. The new section can then be placed anywhere in memory.

MERGE can be used during both incremental and absolute links.

MERGE commands will be executed in the order that they are found in the command file.

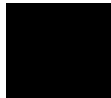
---

#### Note

MERGE and ALIAS are mutually exclusive. Any attempt to use both commands in the same session will result in an error.

**Example:**

```
; There are three modules each containing three
; sections: SECT1, SECT2, SECT3.
;
MERGE NEW_SECT SECT1,{SECT2,MOD2},{SECT3,MOD3}
MERGE NEW_SECT {SECT3,MOD2}
;
SECT NEW_SECT=$1000
SECT SECT2=$2000
SECT SECT3=$3000
;
LOAD MOD1,MOD2,MOD3
;
; This causes a new section with the name NEW_SECT to
; be created. It is located at $1000
; containing the following module sections in the order listed:
; SECT1/MOD1, SECT1/MOD2, SECT1/MOD3, SECT2/MOD2, SECT3/MOD3,
; SECT3/MOD2.
;
; There is also SECT2 located at $2000 containing:
; SECT2/MOD2, SECT2/MOD3 and
; SECT3 located at $3000 containing:
; SECT3,MOD1
```



**NAME**

---

**NAME**

**Specify Output Module Name**

**Syntax:**

Command	Argument
NAME	name

**Where:**

name                    A symbol that specifies the object module name.

**Description:**

The NAME command is used to specify the name of the final output object module. This appears on the first line of the output object file as an extension to the standard Motorola S-record hexadecimal format, which does not contain a name. Any symbols assigned values by the PUBLIC command are considered to lie in this load-time-defined module.

The name may be any standard symbol, up to 31 characters long. If you do not specify a name, the name of the output module will be taken from the first input module.

**Example:**

```
NAME            READER
```

---

## NLIST

### Clear Loader Options

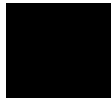
**Syntax:**

Command	Argument
NLIST	FLAG {,FLAG} . . .

**Where:**

FLAG is one of the following:

- A Specifies the output file format to be Motorola S-Record. (Default= IEEE.)
- C Specifies that no cross reference listing is to be produced. (Default= NLIST C)
- I Specifies the output file format to be IEEE. (Default= IEEE.)
- O Specifies that no object module is to be produced. (Default= LIST O)
- P The P flag only affects Motorola S-Record output files. It specifies that the local symbols from input modules loaded (while this flag is set) be included in the output file. This flag can be turned off and on between LOAD commands. Its purpose is to exclude local symbols from particular modules because of duplicate symbol conflicts. (Default= LIST P)
- S Specifies that no local symbol table information is to be written to the output file. (Default= LIST S)
- T Specifies that the local symbol table is not listed on the loader listing. (Default= NLIST T)
- X Specifies that the PUBLIC (global) symbol table is not to be listed on the loader listing. (Default= NLIST X)



## Linker/Loader Commands

### NLIST

**Description:** The NLIST command is the opposite of the LIST command and is used to suppress the listing of the elements specified. The elements may be turned back on with the LIST command.

**Example:**

```
NLIST      0          ; don't produce an  
           0          ; object module
```

---

## **NOPAGE**

### **Turn Off Page Relocatability**

**Syntax:**

Command	Argument
NOPAGE	sname

**Where:**

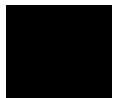
sname            A section name, which should have previously appeared in a PAGE directive.

**Description:**

This directive restores the Relocation Type of a section to Word. It is legal but unnecessary unless the specified section has previously appeared in a PAGE command.

A section name appearing here for the first time is assigned the long attribute but is assigned neither the common nor the noncommon attribute.

The typical use for this command is to turn off paging for modules which are already known to work correctly (libraries, for instance), in order to save memory space.



## **ORDER/SORDER**

### **Specify Long/Short Section Order**

**Syntax:**

Command	Argument
ORDER	lname{(sect_type)}{,lname{(sect_type)}}...
SORDER	sname{(sect_type)}{,sname{(sect_type)}}...

**Where:**

lname	The name of a section with the long attribute.
sname	The name of a section with the short attribute.
sect_type	Specifies a section type. Section type can be C for code, D for data, M for mixed, or R for ROM data.

---

**Note**

The same section name may not appear twice on an ORDER or SORDER command. Multiple ORDER or SORDER commands are accepted without a warning and concatenated.

---

**Description:**

These commands alter the default order of assigning Load Addresses to sections.

As described in the "Linker/Loader Operation" chapter, the normal order of the sections in each group (the groups are [I] short sections and [II] long sections) is just the order in which the loader encountered their names. Use the ORDER and SORDER commands when you do not need to specify load addresses for each section but would like the sections to be placed in memory in a different order.

If you specify load addresses for the sections, the order of the sections might not be important. Keep in mind, however, that even if a load address is specified for a certain section, any sections assigned memory space after that section will be loaded at the next available address. If, for example, you want long section SECT2 to begin at \$FFFF00, and all the other long sections to be



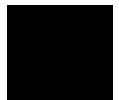
placed together in lower memory, the **ORDER** command should be specified with section **SECT2** being the last argument. If this is not done, then any sections which are listed after **SECT2** will reside in upper memory above section **SECT2**.

If you ask the loader to place one section where it would overlap another, the loader will place the section at the first address which avoids the overlap, even if that means changing the order from what you specified.

While the information necessary to determine the default order of the sections is available to you, in complex cases it will be simpler to use the **ORDER** command than to figure out the default order. The **ORDER** command applies to long sections; the **SORDER** command to short sections.

If a section name appears in these commands for the first time it is assigned the appropriate shortness attribute, but it is assigned neither the common nor the noncommon attribute so that subsequently it may turn out to be either. If the name of a short section appears in the long version of the **ORDER** command this is a fatal error; however the final determination of which sections are short cannot be made until all modules have been read, since any short subsection declaration makes a section short. If the name of a long section appears in the short version of the **ORDER** command a warning is printed and the section is given the short attribute. (This may occur if a **SECT**, **COMMON**, **PAGE**, **CPAGE**, or **NOPAGE** directive precedes the **SORDER** command, since these directives assign newly found sections the long attribute.)

Specify the order of the sections within each group by specifying section names separated by commas. Any sections remaining within the group will be assigned memory space after the sections specified in the command in the order their names were encountered by the loader.



Linker/Loader Commands  
**ORDER/SORDER**

**Example:**

```
ORDER      SEC1 , COMSEG  
SORDER    SEC2 , SHORTSEC
```

An ORDER or SORDER command may be continued to the next line by terminating it with a space followed by a pound sign (#). This character must go between section names, like a comma.

```
ORDER      SECT1 #  
           SECT2 , SECT3 #  
           SECT4
```

## PAGE

### Set Noncommon Section to be Page Relocatable

**Syntax:**

Command	Argument
PAGE	sname

**Where:**

sname            A section name.

**Description:**

This command may be used to modify the relocation type of a noncommon section(s) in the input object modules to Page. As explained in the section titled Relocation Types, all sections are assumed to be Word Relocatable at first. This command allows you to override the default relocation type. After the PAGE command is read, each subsection of the specified section loaded thereafter will be loaded at the next nearest 256 byte boundary until a NOPAGE command for the section is encountered.

The typical use of this command is to allow you to begin each section on a page boundary, for ease of debugging. After debugging is completed the Page commands are removed to avoid wasted memory space.

Specify the section name. If this is the first occurrence of this section name it is given the attributes noncommon and long.

**Example:**

```
PAGE        SECT1
PAGE        SECT2
```



## **PUBLIC**

### **Specify Public Symbols (External Definitions)**

**Syntax:**

Command	Argument
PUBLIC	sym= value
PUBLIC	sym= sym2{ + offset }
PUBLIC	sym= sym2{ -offset }

**Where:**

sym	A user defined external definition symbol.
value	A constant number.
sym2	Another global sym defined in a module or in a previous PUBLIC command.
offset	A constant value that may be added or subtracted from symbol 2.

**Description:**

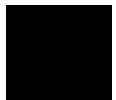
This command is used to define and/or change the value of an external definition (XDEF). Symbol names specified by the loader's PUBLIC command take precedence over symbol names defined during assembly. Therefore, if a symbol specified by this command is already an external definition (from an input object module defined by the assembler), the value of the symbol is changed to that specified in the PUBLIC command. If the symbol is not already defined, it will be entered into the loader's symbol table along with the specified value and will then be available to satisfy external references from object modules. Symbols defined in the PUBLIC command are absolute if their definition is a number or another absolute symbol; they are relocatable if defined as equal to a relocatable symbol.

This command is position dependent in the linker command file. Public symbols defined before a library containing the symbol is loaded with not be resolved properly.

This command allows you to specify the value of some external symbols at Load time and possibly to avoid a reassembly.

**Example:**

```
PUBLIC    INPUT=$2F
PUBLIC    OUTPUT=$200
PUBLIC    newsymbol=oldsymbol
```



## **RESADD/RESMEM**

### **Reserves Regions of Memory**

**Syntax:**

Command	Argument
RESADD	low_addr,high_addr
RESMEM	low_addr,size

**Where:**

low_addr	Starting address of the memory to be reserved.
high_addr	Ending address of the memory to be reserved.
size	Number of bytes to be reserved.

**Description:**

The RESADD/RESMEM commands reserve specified memory locations. The reserved memory region is made into an absolute section that will show up in the SECTION SUMMARY of the link map.

When sections are placed using ORDER or SORDER commands, nothing will be loaded in the reserved memory region. This can be useful for "skipping" a region of memory for a real-time operating system, for example.

If a section is placed at a specific address using the SECT command, and the section overlaps a reserved region, a non-fatal error message is issued. The load will still continue to completion, but the resulting absolute file will contain sections at overlapping addresses. The linker issues a warning if high\_addr is less than low\_addr for RESADD.

RESADD reserves the addresses low\_addr to high\_addr. RESMEM reserves the addresses low\_addr to low\_addr + (size-1). The low\_addr, high\_addr, and size are all numeric constants.

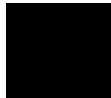
**Example:**

This command file:

```
RESMEM $200,$100
RESADD $2,$101
LOAD module1
END
```

will generate the following two entries in the SECTION SUMMARY of the resulting link map if there are no overlapping sections.

```
SECTION SUMMARY
SECTION ATTRIBUTE  START      END        LENGTH    ALIGN
ABSOLUTE          00000200  000002FF  00000100  0 (BYTE)
ABSOLUTE          00000002  00000101  00000100  0 (BYTE)
```



## **SECT**

### **Set Noncommon Section Load Address**

**Syntax:**

Command	Argument
SECT	sname,value
SECT	sname= value
SECT	sname value

**Where:**

sname	Specifies the section name.
value	Specifies the load address of the section.

**Description:**

The SECT command is used to specify the load address of a noncommon Relocatable section. If this is the first occurrence of the section name, it is given the attributes noncommon and long. Any use of this command must precede any LOAD commands.

Specifies the section name followed by the address of the location at which to start loading the section. The specified address will be rounded up to the next alignment boundary in all cases, and to the next page boundary if paging is in effect for the first subsection of the section.

**Example:**

```
SECT    SECT1 , $400
SECT    SECT2=$1320
```

---

**Note**

The value is separated from the section name by a blank, comma, or equal sign. Multiple SECT commands with the same section name are accepted without a warning, but only the last one will be used.



---

## SECTSIZE

### Set Minimum Section Size

#### Syntax:

Command	Argument
SECTSIZE	sname= size {,sname= size }...

#### Where:

sname	Specifies the section name.
size	Specifies a constant representing the minimum section size, in bytes.

#### Description:

The SECTSIZE command specifies the minimum size in bytes of a combined continuous memory space defined by *sname*. It is an error to define a size less than the size of the combined section unless the section is of type common.

If the section does not exist, it will be created and considered to be noncommon.

#### Example:

```
SECTSIZE    STACK=$100
```

---

#### Note

The value is separated from the section name by a blank, comma, or equal sign. Multiple SECT commands with the same section name are accepted without a warning, but only the last one will be used.



## **START**

### **Specify Output Module Starting Address**

**Syntax:**

Command	Argument
START	value

**Where:**

value                      Specifies the starting address to be used in the output object module.

**Description:**

This command is used to specify the absolute starting address to be placed in the terminator record of the object module. If not specified, the starting address is obtained from the END record of the main program of the input modules. If no main program has been read, the starting address will be zero.

Evidently this directive should not be used unless the starting address falls in an absolute section or in a relocatable section with a specified load address. In the latter case, be warned that when the load address is rounded upwards to lie on a word or page boundary, the starting address is not likewise rounded.

**Example:**


```
START        $7FC
```



---

## Librarian Introduction

This chapter describes the operation of the ar68k librarian.



The ar68k object module librarian may be used to build program libraries, which are collections of relocatable object modules residing in a single file. These libraries enable you to load frequently used object modules by referring to publicly defined names, without concern for the specific names and characteristics of the modules. The librarian accepts the relocatable object module output of the as68k assembler.

The librarian performs the function of formatting and organizing library files that will subsequently be used by the ld68k linking loader. Libraries are both a convenient means for managing collections of relocatable object modules and a more efficient means for linkers to access the modules when required. This efficiency is realized by reducing the number of files that must be opened for linking modules.

The word "module", as used in discussing the librarian, refers to a Relocatable Object Module that results from assembling a source program, using the as68k Relocatable Macro Assembler. Modules in a library must be in the format produced by the Assembler.

This, and subsequent chapters, describe the ar68k librarian that accompanies the as68k assembler, how to build and manipulate the libraries, and how the ld68k loader utilizes the libraries.

---

## **Librarian Features**

The ar68k Object Module Librarian features the following:

- User friendly commands.
- Efficient operation.
- Batch Command line input and return codes for "make" type procedures.
- Optimized structure for fast linker access.

## **Librarian Operation**

The librarian may be utilized in both an interactive or a batch mode. In interactive mode, you interact with the librarian directly by entering commands and receiving status responses. All commands are available in this mode. There are two types of batch input modes available. The first uses an input command file which can contain any of the available commands and outputs resulting status messages to a listing file. The second batch mode uses input commands from the command line only. This mode is limited to addmod, delete, extract and replace functions within an existing library.

---

## **Librarian Function -- Overview**

When writing modular programs (using Relocatable Macro Assemblers), communication among the various modules is established through use of XDEF and EXTERNAL Symbols. For example, the following illustration shows three relocatable object modules that resulted from the assembly of generic assembly language modules.

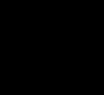
## Librarian Introduction Librarian Function -- Overview

```
SQUARE      IDNT
             XDEF      NEXT
             XREF      FALLOW

;           Program Module 1

             JMP      FALLOW
NEXT:       NOP
             NOP
             END
```

A Relocatable Object  
Module that resides in  
host system file  
"knewel.o".



```
SINCOS      IDNT
             XDEF      FALLOW
             XREF      NEXT

;           Program Module 2

FALLOW:     MOVE      #8,D0
             NOP
             JMP      NEXT
             END
```

A Relocatable Object  
Module that resides in  
host system file  
"swigget.o".

```
ARCTAN      IDNT
             XDEF      ARCTAN

;           Program Module n

ARCTAN:     MOVE      260,D1
             MULS     D1,D2
             NOP
             END
```

A Relocatable Object  
Module that resides in  
host system file  
"bayer.o".

Of the three modules shown, the first two can be seen to communicate with one another through external references and public symbols, while the third is a stand-alone module.

Librarian Introduction  
**Librarian Function -- Overview**

The relocatable modules illustrated consist of load data information, relocation information, and records that indicate:

- 1) Public symbols
- 2) External symbols.

The above relocatable object modules may be made Members of a library by various combinations of librarian commands.

For example, a new library may be created by the following command. Substitute ".lib" for ".a" if you are using a DOS system.

```
CREATE newrem.a  
SAVE
```

or:

```
CREATE newrem.a  
ADDMOD knewel.o  
ADDMOD swigget,bayer  
SAVE
```

There are several ways that the relocatable object modules can be incorporated into a library by utilizing various librarian commands, which are described in detail in the "Librarian Commands" chapter. Now that a library containing these members has been built, it may be used by the ld68k loader.

Assume that you have written a program called "main". After "main" has been assembled, the Relocatable Object Module that results is in a host system file named "main.o" (or "main.obj" for DOS systems). This module has a reference to the public symbol ARCTAN.

```
XREF    ARCTAN  
  
Main Module  
NOP  
JSR     ARCTAN  
NOP  
END
```

```
A Relocatable Object  
Module in host system  
file "main.o".
```



Before the existence of the library, you could have directed the loader as follows.

```
LOAD  main.o
LOAD  bayer.o
```

Now that there is a library, you can direct the loader as follows.

```
LOAD main.o
LOAD newrem.a
```

The linking loader will access the library to attempt to resolve external references, such as ARCTAN. Now, if we modify the "main" module so that it calls the SINCOS module as well:

```
XREF  ARCTAN
XREF  FALLOW
Main Module
NOP
JSR   ARCTAN
JSR   FALLOW
END
```

A Relocatable Object Module in host system file "main.o".

Without the ability to load from a library, it would be necessary to command the linking loader as follows.

```
LOAD  main.o
LOAD  swigget.o
LOAD  bayer.o
LOAD  knewel.o
```

## Librarian Introduction

### Librarian Function -- Overview

However, when using a linking loader with the ability to load from a library, you need specify only:

```
LOAD    main
LOAD    newrem.a
```

The loader will load the relocatable object module "main" in the usual way. It will load the other modules from the library.

The following is a more practical example of the use of the library.

A programmer writes a series of program modules consisting of a number of mathematical routines including a few modules that calculate transcendental functions. Then, these modules are gathered into a library file, through use of the ar68k librarian.

Sometime later, a programmer, either the one who wrote the mathematical routines or someone else, has a requirement to calculate an Arc-Tangent function within a program he is writing. He is aware of the fact that there is an Arc-Tangent Function in a library file. He knows the name of the Entry Point of the routine and he also knows how to pass parameters to the Arc-Tangent Function and how to accept the result of the calculation.

So, during the coding of his program he need do only two things:

- 1 JSR the Arc-Tangent function from the program he is developing, placing the Public Name of the Entry Point into the argument field of the JSR or JMP instruction.
- 2 Place the Public Entry Point name of the Arc-Tangent Function in the argument field of an External Reference Pseudo-Op in the program he is writing.

Even though he does not know the name of the relocatable object module that contains the Arc-Tangent Function, he will be able to direct the linking loader to include the relocatable module that contains the correct module simply by informing the loader to use the required library file(s).

The ld68k linking loader need not be explicitly informed which module contains the Arc-Tangent Function. The loader will automatically search the named library, looking for the Entry Point name that the programmer wrote as the argument of his JSR statement. When the Entry Point name has been found, the loader identifies the module in which it resides, and then includes the module containing the name in the current load.

The loader determines which of the library modules to load by examining the internal list of unresolved external references that it accumulated during the load process and then accessing the library file to determine if there is a match between such an unresolved external reference and a label or name that has been declared Public in one of the modules in the library file. The loader then identifies which module contains the matching Public symbol and loads it just as if he had explicitly directed the loader to load the proper module.

Even if there are several unresolved external references, the loader will attempt to load every module that contains corresponding public symbols, in order to satisfy every possible reference. Even when the inclusion of a module in the library adds an undefined reference to the list of undefined references, the load will access the library again until all external references have been satisfied. All public symbols within a library must have unique names.

The advantages of using a library are as follows.

- A user need only know the input parameters, output parameters, and entry point name of the function in order to have it included in the final load module.
- A library that is a collection of often-used functions can reside on your system and be available at all times to everyone.
- Module names and entry point names of all the program modules you create are easy to track.

## Command Syntax

The librarian recognizes six special characters:

*	-	asterisk
;	-	semicolon
,	-	comma
(	-	left parenthesis
)	-	right parenthesis
+	-	plus

### Use of Special Characters

The use of these special characters in the command syntax is described below.

Filename implies the ordinary file name syntax that would be used on the host system.

Module names are written according to the rules for the assembler used to create the Relocatable Object Modules. Each module must have a unique module name.

Public symbols are written according to the definition given in the assembler used to create the modules.

The asterisk (\*) and the semicolon (;), when appearing in a Command line, cause the librarian to ignore the rest of the line.

These characters may be used to place comments in a command sequence. The librarian does not process the rest of the line, which will be written to the output file as a comment.

The comma (,) separates members of a list of similar elements. The list may contain module names, or module filenames.

The left and right parentheses (), used in pairs, denote a list of similar elements in a command. Parentheses may be used to group module names that are members of a library only.

The plus sign (+) followed by a carriage return allows you to continue a list on subsequent line(s). Care should be exercised when using it. Do not break up or interrupt a complete syntactical unit (i.e., do not try to continue a filename, a module name, or a command). The command verb must be terminated by a blank if it was an argument. If the continuation character (+)

is used immediately after the command verb, it must be separated from the command by at least one blank.

Except as noted above, the line continuation character may appear anywhere in a command.

## Blanks

Except as noted above, blanks may be used freely within commands (between syntactically identifiable units).

Example:

```
DELETE  MOD1 , MOD2
```

is the same as:

```
DELETE MOD1,MOD2
```

## Command File Comments

Comments may be included in a command file to document the processing. These are included by use of the semicolon (;) or asterisk (\*).

Example:

```
;      this is a complete line of comment  
addmod modulea.o ; this is a command line comment  
addmod moduleb      * this is another comment
```

## Module Names

A module is the output generated when assembling source files. The module name is controlled by the IDNT directive. If no IDNT directive is specified in the assembly source file, then the module name is the source file name with any leading path or trailing suffix stripped. If an IDNT directive is specified, the module name is taken from the IDNT directive.

## **Return Codes**

The librarian returns 0 if no errors are detected; otherwise, it returns nonzero. The librarian will complete normally, issue an informative message, issue a warning, or end abnormally with an error. Error messages and warnings are listed in the "Librarian Error Messages" appendix.

---

## **Library Listing Format**

The output listing contains the following information:

- Header information including the time of the library creation and the version number of the librarian.
- A list of the librarian commands and the name of the library in progress.
- A list of the modules contained in the library. The public symbols defined and the external symbols used in each module are listed, as well as a count of the public and external symbols.
- A count of the number of modules in the library.

---

## **Sample Test Program Description**

The sample test programs in the next section utilize command files and object module files. If the object module files on your disk have different names, you must edit the command file and replace these assumed names with the actual names.

When you start the librarian, it will display a header and a prompt character. Commands can be entered at this point. The librarian can also be run in batch mode.

The following pages show the results of a librarian sample test execution. The information is displayed at the terminal during interactive program execution. If in batch mode, the information is printed in an output stream formatted similar to those appearing in the next section.

---

## Example Librarian Listing



```
HPB3641-19300      Wed Apr 28 15:19:56 1993

  Version A.02.00
* Create a library called "exlib.a", add two relocatable
* modules, get a brief listing and a complete listing,
* save the current library, and exit.

CREATE exlib.a
ADDMOD transfer.o
ADDMOD delay.o
DIRECTORY exlib.a exlib.dir
LIST exlib.a
HPB3641-19300      A.02.00 Wed Apr 28 15:19:57 1993

Library being built exlib.a

  Module          Size  Processor
transfer ...      352   68000

  ***** PUBLIC DEFINITIONS *****
TRANSFER

  ***** EXTERNAL REFERENCES *****
VIDEO_RAM

Public Count      = 1
External Count    = 1

  Module          Size  Processor
delay ...         307   68000

  ***** PUBLIC DEFINITIONS *****
DELAY

Public Count      = 1
External Count    = 0
Module Count      = 2
SAVE
END
```

Figure 11-5. Example Librarian Listing

### Description of Example

In this sample program, a new library, exlib.a, is created and two modules (transfer.o and delay.o) are added to it without error. (If you are using a DOS system, the extensions will be ".lib" and ".obj".) The contents of the library are then listed in an output stream. The output stream shows each Module Name, and its public definitions and external references during execution. Symbols are case sensitive. The total public symbol count and total external symbol count are listed for each module. The total module count as well as total warnings and errors are displayed at the end of the output stream.

---

## Brief Format Example Library Listing

### Brief Format Listing Description

HPB3641-19300 A.02.00 Tue Apr 27 15:44:36 1993

Library being built exlib.a

Module	Size	Processor
transfer ...	352	68000
delay .....	307	68000

Module Count = 2

### Figure 11-6. Brief Format Example Library Listing

The brief format library listing shown above was generated with the DIRECTORY loader command shown in the first listing. The name of the library, the names of the modules in the library (and their sizes), and the module count are included in the brief format listing.



---

# 12



---

## Librarian Commands

This chapter describes the commands that are used by the ar68k Object Module Librarian.

The Librarian reads a sequence of commands from the command input device. Commands may be read in interactive or batch mode. The command sequence must be terminated by an END or SAVE command. Relocatable object modules are read as input and collected in organized libraries as specified in the command input file.

---

## Command Summary

The following list summarizes the commands described in this chapter.

ADDLIB	Include Library Object Module in Current Library.
ADDMOD	Add Object Module to Current Library.
CLEAR	Remove the Current Library.
CREATE	Define New Library.
DELETE	Delete Module From Current Library.
DIRECTORY	Brief Listing of Contents of Library.
END, EXIT, QUIT	Terminate Execution of Librarian.
EXTRACT	Copy Library Module to File.
FULLDIR, LIST	List Contents of Library or Library Module.
HELP	Display Current Valid Commands and Syntax.
OPEN	Open an Existing Library.
REPLACE	Replace Library Module.
SAVE	Create Library File Saving Contents of Current Library.

## ADDLIB

### Include Library Object Module in Current Library

**Syntax:**

Command	Argument
ADDLIB	{path}libname{(mod{,mod} . . .)}

**Where:**

path	Host specific path specification.
libname	Library filename from which to add module(s). If the library filename specified has a suffix, the name is used as is. If the library file name specified has no suffix, the suffix ".a" (".lib" for DOS) suffix is appended to "libname" before it is used.
mod	Name of relocatable object module(s) to include; if none are specified, the entire library is included.

**Description:**

The ADDLIB command is used to specify that object modules from another library are to be included in the library currently being created or modified. An OPEN or CREATE command must precede the ADDLIB to open or create the library to which the modules will be added.

**Example:**

```
ADDLIB MATH.a (SQUARE,SQROOT)
```

The above Command directs the librarian to include the "SQUARE" and the "SQROOT" Modules from library named MATH.a. into the current library.

## **ADDMOD**

### **Add Object Module to Current Library**

**Syntax:**

Command	Argument
ADDMOD	filename {,filename} . . .

**Where:**

filename           Filename (including path) of file containing the Relocatable Object Module to be added to the library. If the filenames contain suffixes, the filename is used as is. If the filenames have no suffixes, then ".o" (or ".obj" for DOS) is appended to the filename before it is used.

**Description:**

The ADDMOD command specifies that an object module that is not in a library file is to be included in the library currently being created or modified. The module(s) to be added to the library should have been named with the NAME directive at assembly time. The ADDMOD command must be preceded by an OPEN or CREATE library command.

**Example:**

```
ADDMOD MATH.MBR
```

The above Command directs the librarian to add a Relocatable Object Module from file "MATH.MBR" to the current library.

---

## **CLEAR**

### **Erase the Current Library**

#### **Syntax:**

Command	Argument
CLEAR	

#### **Description:**

Clears all library commands that have been entered in the current session since the last SAVE command. Another CREATE or OPEN may then be issued. This command is useful if you access several libraries in a single librarian session.

#### **Example:**

```
OPEN    lib1
DIR     lib1
CLEAR  ; allow a new current library
OPEN    lib2
```

## **CREATE**

### **Define New Library**

**Syntax:**

Command	Argument
CREATE	{path}libname

**Where:**

path	Host specific path specification.
libname	Library filename. If the name of the library file has a suffix, then it is used as is. If the library file name has no suffix, then ".a" (or ".lib" for DOS) is appended to create the library file name.

**Description:**

The CREATE command specifies the name of a new library which becomes the current library for the remainder of the commands.

**Example:**

```
CREATE TEMPOR.a
```

The above Command directs the librarian to create a file, "TEMPOR.a", on the host system and format it as a library. If the file TEMPOR.a already exists, the user will be given a warning in interactive mode. In batch mode, no library will be created.

---

## **DELETE**

### **Delete Module From Current Library**

**Syntax:**

Command	Argument
DELETE	mod{,mod} . . .

**Where:**

mod                      Name of module(s) to be removed from library named in preceding OPEN or CREATE command.

**Description:**

The DELETE command is used to specify module(s) to be removed from the library currently being created or updated. The module names specified are the Relocatable Object Modules that are to be deleted. The module name may be defined with the IDNT or the NAME assembly directives. If IDNT or NAME are not used, the module name is the name of the assembly source file, with any preceding path or trailing suffix stripped.

**Example:**

```
DELETE ARCTAN,SQUARE,RAD
```

The above Command directs the librarian to delete the "ARCTAN", "SQUARE" and "RAD" relocatable object modules from the current library.

## **DIRECTORY**

### **Brief Listing of Library Contents**

**Syntax:**

Command	Argument
DIRECTORY	{path}libname{(mod{,mod} . . .)} {listfile}

**Where:**

path	Host specific path specification.
libname	Library file referenced; the current library is referenced by its name. If the name of the library file has a suffix, then it is used as is. If the library file name has no suffix, then ".a" (or ".lib" for DOS) is appended to create the library file name.
mod	Module to be listed.
listfile	Filename to receive listing; if not specified, default to standard output (usually the terminal).

**Description:**

The DIRECTORY command is used to request a brief listing of the contents of a library. The directory listed is of the library specified by the user. The user may specify the current library or another library. All modules in the library are listed with their Module Names and Module sizes (in bytes).



**Example:**

```
DIRECTORY SIEVE.a (command input)
```

```
Library SIEVE.a
```

```
  \
Name  Size \
SIEVE ... 1812 \
MODULE ... 228 (output)
MODULE1 .. 1032 /
  /
```

```
Number of Modules = 3
```

The above DIRECTORY Command will produce the listing of the modules in SIEVE.a and the size (in bytes) of each module as shown. The listing is produced on the standard output.



## **END, EXIT, QUIT**

### **Terminate Execution of Librarian**

**Syntax:**

Command	Argument
END	
EXIT	
QUIT	

**Description:**

The END command (and variations) is used to terminate command processing in the librarian. The END command does *not* cause the current library to be saved. The results of previous commands are *not* saved. In order to save the current library, you must terminate using the SAVE command.

**Example:**

```
ar68k
LIST NEW.a
END
```

In this example, the librarian program is opened so that the user may list the contents of library NEW.a. The librarian is exited using the END command as soon as the information needed has been received.

---

## EXTRACT

### Copy Library Module to File

**Syntax:**

Command	Argument
EXTRACT	mod{,mod} . . .

**Where:**

mod                      Name of module to be copied. The name of the output file is the module name with ".o" (or ".obj") appended.

**Description:**

The EXTRACT command is used to specify a library module that is to be copied to a non-library file. EXTRACT is the converse of the ADDMOD command. This command directs the librarian to copy the specified library module, which is a catalogued member of the library file, out to an external file in the host system. The extracted module is in the same format as when it was generated by the Assembler; consequently, it can be loaded explicitly by the Linking Loader.

The EXTRACT command must be preceded by an OPEN or CREATE command for the library from which the extract is to occur.

**Example:**

```
EXTRACT MODA,MODB,MODC
```

In the above example, a list of modules is specified. The modules MODA, MODB, and MODC are copied from the current library into object files of the same names, but with ".o" appended. The filenames created in this case are MODA.o, MODB.o, and MODC.o. The extension may be different on your operating system. Refer to page 25 for a list of filename extensions on your operating system.

## FULLDIR, LIST

### List Contents of Library or Library Module

#### Syntax:

Command	Argument
FULLDIR	{path}libname{(mod{,mod} . . .)} {listfile}
LIST	{path}libname{(mod{,mod} . . .)} {listfile}

#### Where:

path	Host specific path specification.
libname	Library file referenced. If the name of the library file has a suffix, then it is used as is. If the library file name has no suffix, then ".a" (".lib" for DOS) is appended to create the library file name.
mod	Module to be listed.
listfile	Filename to receive listing; if not specified, default to standard list device.

#### Description:

The LIST command is used to request a complete or partial listing of a library. Every specified module is listed, along with a list of External References and Public Symbols. See the "Librarian Listing Description" chapter for the listing format.

#### Example:

```
LIST TRIG.a (ARCSIN,TANGEN) TRIG.LIST
```

The above Command will cause the librarian to write information pertaining to modules "ARCSIN" and "TANGEN", which are members of the library "TRIG.a" into the host system file "TRIG.LIST".

---

## HELP

### Display Current Valid Commands and Syntax

**Syntax:**

Command	Argument
HELP	

**Description:**

The HELP command displays the currently valid librarian commands and the acceptable syntax for each.

The librarian has two contexts with different valid commands. The first context is when there is no current library, in other words, no CREATE or OPEN command has been executed or a SAVE or CLEAR command has been issued. The second context is when a current library exists.

**Example:**

```
ar68k> help
CLEAR
CREATE library_name
DIRECTORY library_name[(module_name[,...])] [list_filename]
END
FULDIR library_name[(module_name[,...])] [list_filename]
HELP
OPEN library_name
SAVE
```

```
ar68k> create lib1
```

```
ar68k> help
ADDLIB library_name[module_name[,...]]
ADDMOD filename[,...]
CLEAR
DELETE module_name[,...]
DIRECTORY library_name[(module_name[,...])] [list_filename]
END
```

Librarian Commands  
**HELP**

EXTRACT module\_name[...]  
FULLDIR library\_name[(module\_name[...])] [list\_filename]  
HELP  
REPLACE filename[,filename]  
SAVE



## OPEN

### Open an Existing Library

**Syntax:**

Command	Argument
OPEN	{path}libname{(module{,module} . . .)}

**Where:**

path	Host specific path specification.
libname	Library file name. If the name of the library file has a suffix, then it is used as is. If the library file name has no suffix, then ".a" (".lib" for DOS) is appended to create the library file name.
module	If modules are specified, only those modules are included in the current library. If no modules are specified, all modules from "libname" are included.

**Description:**

The OPEN command is used to specify that an existing library is to be referenced in conjunction with succeeding maintenance commands. If the maintenance commands require that a new generation of the library be created, the new version or updated library will have the same name as the current library. If the library cannot be located or opened for input, an error is reported. If the librarian is operating in batch mode, execution will be terminated.

**Example:**

```
OPEN MATH.a (ARCSIN,SQUARE)
```

In this example, modules ARCSIN and SQUARE are opened in library MATH.a.

## **REPLACE**

### **Replace Library Module**

**Syntax:**

Command	Argument
REPLACE	filename{,filename} . . .

**Where:**

filename	Filename of file containing module. If the filenames contain suffixes, the filename is used as is. If the filenames have no suffixes, then ".o" (".obj" for DOS) is appended to the filename before it is used.
----------	---

**Description:**

The REPLACE command is used to replace a library module with a non-library module of the same name. This command directs the librarian to open a named module file. The library module is then replaced with a module of the same name from the Non-library file it opened.

**Example:**

```
REPLACE SENTIN.o
```

The Command above directs the librarian to replace module SENTIN with a copy of the module named SENTIN, located in a file named SENTIN.o.



---

## **SAVE**

### **Create Library File Saving Contents of Current Library**

**Syntax:**

Command	Argument
SAVE	

**Description:**

The SAVE command is used to terminate the librarian and write the current library, saving the results of the preceding commands.

Prior to the SAVE command, the maintenance commands preceding were only checked for correct syntax and module existence. At SAVE time the actual processing of the maintenance commands takes place. SAVE indicates that a library is to be built following the rules of the preceding commands.

**Example:**

```
CREATE NEW.a
ADDMOD REL1.o, REL2.o
ADDMOD FORTUN.o
SAVE
```

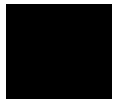
In this example, "REL1", "REL2" and "FORTUN" relocatable object modules will be saved in library named NEW.a.

Librarian Commands  
**SAVE**



---

# A



---

## Assembler Error Messages

This appendix describes the error messages and warnings that appear if errors in the source program are detected during the assembly process.

Assembler Error Messages  
**(500)–(517)**

The error message is printed on the listing immediately following the statement in error.

The following list will serve as a guide to diagnosing the error. Most error messages are self-explanatory. The listing displays a total error count. See the "Error Message Formats" appendix for explanations of error severity levels.

The errors and messages for the Assembler are listed and described below.

- (500) **No error.**
- (501) **Missing argument.**  
The argument is missing or contains an illegal character, etc. Mismatch on common/noncommon section type.
- (502) **Operator expected but not found.**
- (503) **A symbol was found which is invalid in this context.**
- (504) **Right parenthesis not valid in this context.**
- (505) **Operator not valid in this context.**
- (506) **Expression terminator found prematurely.**
- (507) **Operand expected but not found.**
- (508) **Unbalanced parentheses.**
- (509) **Complex relocatable value not valid in this context.**
- (510) **Stack underflow (internal error).**
- (511) **Invalid operands for \' operator.**
- (512) **Invalid operands for & operator.**
- (513) **Invalid operands for | operator.**
- (514) **Invalid operands for || operator.**
- (515) **Invalid operands for = operator.**
- (516) **Invalid operands for < > operator.**
- (517) **Invalid operands for > = operator.**

(518) **Invalid operands for > operator.**

(519) **Invalid operands for < operator.**

(520) **Invalid operands for <= operator.**

(521) **Invalid operands for >> operator.**

(522) **Invalid operands for << operator.**

(523) **Invalid operands for \* operator.**

(524) **Invalid operands for / operator.**

(525) **Invalid character.**

This message is produced as the result of a variety of syntactic errors. A character may be invalid within the context where it is found. The input line may be too long. A register name may be found where one is not allowed.

(526) **Closing string delimiter missing.**

(527) **String longer than 4 characters invalid in this context.**

(528) **Invalid opcode.**

(529) **Invalid opcode/qualifier combination.**

(530) **Undefined symbol.**

There is a symbolic name in the operand field that has never been defined. The symbol should have been previously defined for certain directives and was not, but may have been defined after the directive. A symbol declared on the XDEF directive was not used in the program.

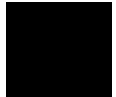
(531) **Invalid nesting of IF . . . ENDC..**

(532) **Invalid nesting of IF . . . ELSEC . . . ENDC.**

The opcode mnemonic is not a valid instruction, directive, or a macro call. A macro defined within another macro, or conditional assembly statements are nested too deeply. ELSEC, ENDC, or ENDM has been used without preceding IF or MACRO.

(533) **Missing ENDC.**

(534) **IF stack overflow; limit is 16 nesting levels.**



Assembler Error Messages  
**(535)–(553)**

- (535)            **This directive not permitted in absolute assembly.**
- (536)            **Code generation not permitted in OFFSET section.**
- (537)            **Integer value is outside of its legal range.**
- (538)            **Label required on this directive.**
- (539)            **Duplicate IDNT directive (ignored).**
- (540)            **Relocatable expression invalid in this context.**

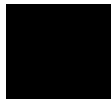
A relocatable expression is used for a field that is not 16 or 32 bits long. An operand that should be absolute is relocatable. An ORG directive makes a reference to an external symbol.

- (541)            **Comma expected but not found.**
- (542)            **Invalid section name.**
- (543)            **Section cannot be both COMMON and non-COMMON.**
- (544)            **Nested macro definition.**
- (545)            **Too many sections.**
- (546)            **Invalid symbol.**
- (547)            **This sort of symbol cannot be made an external definition.**
- (548)            **Invalid external symbol.**
- (549)            **Value will be sign-extended to 32 bits at runtime.**
- (550)            **Unable to open Include file.**
- (551)            **Invalid formal parameter name.**
- (552)            **Invalid local symbol name.**
- (553)            **Duplicate label (ignored).**

The label in the statement has previously appeared in the label field. A label on a SET directive previously appeared in a statement other than a SET directive, or a label on a statement other than a SET directive now appears on a SET directive. A label appears more than once in an XDEF directive. A symbol defined in an XREF directive appears in the label field of some

statement. A keyword appears in the label field or in an XDEF or XREF directive.

- (554) **Incompatible usage: Motorola does not permit a label on this directive.**
- (555) **Section was declared both Short and non-Short. Section will be Short.**
- (556) **NO not permitted on this flag.**
- (557) **Unknown or missing option flag.**
- (558) **Register list invalid in this context.**
- (559) **.W or .L extension on register not valid in this context.**  
User should verify the validity of the extension on register.
- (560) **A register in a colon-separated pair is invalid in this context.**  
Register pairs cannot be separated by a colon in this instruction.
- (561) **A colon-separated pair of registers is invalid in this context.**  
Register pairs cannot be separated by a colon in this instruction.
- (562) **Register expected but not found.**
- (563) **A register in a register list is invalid in this context.**
- (564) **Registers separated by - in register list must be in ascending order.**
- (565) **Registers separated by - in register list must be of same type.**
- (566) **Invalid expression contains a register.**
- (567) **Left parenthesis expected but not found.**
- (568) **Square brackets invalid in this context.**
- (569) **Multiple arithmetic expressions invalid within an operand.**
- (570) **Left brace expected but not found.**
- (571) **Colon expected but not found.**
- (572) **Right brace expected but not found.**
- (573) **Equals sign expected but not found.**

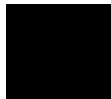


Assembler Error Messages  
(574)–(592)

- (574) **TO or DOWNTO expected but not found.**
- (575) **DO expected but not found.**
- (576) **Nesting of WHILE . . . ENDW invalid.**
- (577) **Nesting of REPEAT . . . UNTIL invalid.**
- (578) **Nesting of IF . . . ELSE . . . ENDI invalid.**  
Nested FILE or INCLUDE directives. ELSE and/or ENDI have been used without the preceding required structural syntax directive.
- (579) **Nesting of IF . . . ENDI invalid.**
- (580) **Nesting of FOR . . . ENDF invalid.**  
Invalid extension for nested FILE or INCLUDE directives. ENDF has been used without the preceding required structural syntax directive.
- (581) **BREAK found outside a structured-syntax loop construct.**
- (582) **NEXT found outside a structured-syntax loop construct.**
- (583) **Invalid condition code in structured syntax directive.**
- (584) **< (condition code) expected but not found.**
- (585) **Code generated is equivalent in some cases. Recoding recommended.**
- (586) **THEN expected but not found.**
- (587) **This instruction has too many operands.**
- (588) **This combination of operands is not valid for this instruction.**
- (589) **Too few bytes allocated on Pass 1 for forward reference.**
- (590) **This instruction will not work on the declared processor type.**  
The instruction or operand is illegal for the specified processor. Use the CHIP directive to specify another processor.
- (591) **FAIL directive assembled.**  
A programmed error has occurred.
- (592) **Register list required for REG directive operand.**



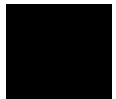
- (593) **This directive invalid outside a macro.**
- (594) **This character invalid within real constant.**
- (595) **A real constant was expected here.**
- (596) **Real numbers invalid in this context.**
- (597) **This real number too small to represent. Zero substituted.**
- (598) **This real number is too large to represent. Infinity substituted.**
- (599) **Macros nested too deeply. Use OPT NEST if this was your intent.**  
When nesting macros, the buffer available for macro parameters is full.
- (600) **Real numbers invalid in this context.**
- (601) **Value was truncated to fit in its field.**  
An evaluated expression or constant is out of range for the field of the actual machine instruction in which it is to be contained.
- (602) **Calculated displacement does not fit in truncated field.**
- (603) **Structured Directives not properly closed.**
- (604) **Local symbols from this section not included in HP asmb\_sym file.**  
When assembling with the "generate HP format output files" option, more than one relocatable section was mapped to HP section PROG, HP section DATA, or HP section COMN. Local symbols from these extra sections are not written to the "asmb\_sym" assembler symbol file and will not be available for debugging. To eliminate this warning, move the extra sections into a new source module.
- (605) **Out of virtual memory.**  
You have exceeded the host system's limit for process size. Try using the **-b** (big) command line option.
- (606) **Invalid Value for alignment, can only be 0, 1, 2, or 4.**
- (607) **End of File inside a macro or repeat definition.**



Assembler Error Messages  
**(608)–(625)**

- (608)           **Expression stack overflow.**
- The expression stack can hold about 45 entities. A single expression, therefore, cannot contain more than 45 entities. An entity is a symbol, an operator, a literal, parentheses, and so on. The expression "a+ b" has three entities. If you must create a single expression that has enough entities to overflow the expression stack, you may be able to circumvent the limit by using EQUs to build the expression from subexpressions.
- (609)           **Value is outside of its legal range.**
- (610)           **Illegal branch to odd address.**
- (611)           **Unable to create or open intermediate file.**
- (612)           **Illegal high-level debug syntax.**
- (613)           **Incompatible processor/ co-processor combination.**
- (614)           **User label conflicts with register name.**
- (615)           **Floating point hex number too big for specified size.**
- (616)           **Too many relocations in this section. Limit is 64K.**
- (617)           **(Not Used)**
- (618)
- (619)
- (620)
- (621)           **Macro/repeat definition terminated by assembler.**
- (622)           **Macro expansion buffer overflowed. Truncated.**
- (623)           **Too many formal parameters. Limit is 36.**
- The limit to the number of parameters for a macro is thirty-six. Reduce the number of formal parameters in the macro definition.
- (624)           **Macro names cannot contain a period (.).**
- Periods are not allowed in macro names (except as the first character).
- (625)           **Macro definition has too many local symbols.**
- The maximum number of local symbols allowed in a macro definition is 90.

- (626)           **Invalid model parameter.**  
The model parameter may be missing in the IRP assembler directive.
- (627)           **Expanded macro line is too long.**  
Break the line into two shorter lines.
- (628)           **Recursive expression evaluation.**
- (629)           **Illegal CHIP identifier.**
- (630)           **Invalid operand for .STARTOF. operator**  
Check that the operand is a section name.
- (631)           **Invalid operand for .SIZEOF. operator**  
Check that the operand is a section name.
- (632)           **The number of nesting levels for macros cannot exceed 100.**
- (633)           **.W or .L extension on cache not valid in this context**  
The extension is not allowed on cache registers.
- (634)           **Extra operand(s) ignored**

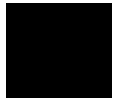


Assembler Error Messages  
**(634)–(634)**



---

# B



---

## Loader Error Messages

This appendix describes the error messages and warnings that may appear during Linking.

## Loader Error Messages

---

Errors and messages from the Loader will be non-fatal or fatal. If the error is non-fatal, the Load will proceed after the error is reported. If the error is fatal, the Loader will report the error, and the load will terminate immediately.

Command errors are usually due to invalid commands or command parameters and usually cause termination of the loading process in batch mode. If command errors are encountered in interactive mode, the Load usually continues.

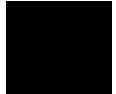
Errors and messages are listed beneath the actual command in error. Load messages normally occur during the loading of object modules initiated by the LOAD command. These messages may be fatal or informative. For most load messages, the message is followed by the record number in the input module and the actual record in error. For a particular module the module name is also listed at the start of the messages.

The mode of operation determines whether the informational message is flagged as a warning or as an error. The severity of the error also varies depending on the mode and environment. In general, the error or message is more severe for the user of a batch file or command line mode, and less severe for the user of interactive mode.

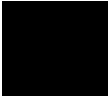
Most load errors should not occur. If they do, the user is advised to first reassemble the program, and then to reload. If the error persists, the user may contact Hewlett-Packard.

The errors and messages are listed and described below.

- (300) **Bad IEEE Object Record.**  
Either the object module has been corrupted or it is not a IEEE relocatable object file.
- (301) **Maximum Number Of Sections Exceeded.**  
The maximum number of allowable sections (2000) has been exceeded.
- (302) **Section Mismatch.**  
A section was typed common in one place and noncommon in another, or short in one place and long in another. This message may arise if a section is mentioned for the first time in a SECT, COMMON, PAGE, CPAGE, or NOPAGE command, as these commands assign the long attribute to newfound sections.
- (303) **Section Overlap.**  
Due to user specified addresses, or absolute sections, one or more of the sections overlap. Some sections of memory may have multiple values loaded. This message is non-fatal and loading continues, but it usually means that you should change the load addresses so the sections do not overlap.
- (304) **Module Too Large.**  
At final load time the combined lengths of all program sections exceed the maximum memory size, established by the CHIP command.
- (305) **Reserved Memory Table Full.**  
There are too many non-adjacent sections in the link. Try to reduce the number of non-adjacent sections.
- (306) **Out of memory.**  
The loader has run out of memory in the host system.
- (307) **Duplicate Public.**  
A PUBLIC is defined that was already defined in another module. Loading will continue and the symbol will be listed.



Loader Error Messages  
**(308)–(318)**

- (308)           **Invalid CHIP Command.**  
The CHIP command as specified by the user is not a legal loader command.
- (309)           **Invalid Command.**  
A command specified by the user is not a legal Loader command.
- (310)           **Load Completed.**  
Message indicates normal load.
- (311)           **Load Not Completed.**  
Message indicates abnormal load.
- (312)           **Invalid ORDER command.**  
The ORDER command specified by the user is not a legal loader command.
-  (313)           **Invalid Operand.**  
An operand specified for a command contains invalid characters, does not exist, or is too large.
- (314)           **Chip inconsist.**  
The loader has encountered a file assembled with a CHIP directive which has "greater" capabilities than the CHIP specified to the loader. For example, a file assembled with the "CHIP 68020" directive is loaded with the "CHIP 68000" load command in effect. The module MAY contain instructions which cannot execute on the target chip.
- (315)           **Maximum memory has been exceeded.**  
The program exceeds the memory available for the target microprocessor.
- (316)           **Short memory has been exceeded.**  
The short memory specified is not enough for all short sections.
- (317)           **Section assigned address below BASE.**  
An absolute or relocatable section has been assigned an address less than the address specified in the BASE command.
- (318)           **Internal Error.**



The loader has encountered a fatal internal error.

(319) **Cannot Open File.**

The loader is unable to open the relocatable object file.

(320) **Unresolved Externals:.**

The unresolved external symbols are listed following this warning message.

(322) **8-bits Value Out of Range.**

A relocated 8-bit value is out of range. An 8-bit field, generally an immediate value, has too large a value. Loading continues but the loaded program often will not run; the user should investigate.

All values are evaluated as unsigned 32-bit values. These values are expected to be within 8 bits sign-extended (i.e., \$FFFFFF80 to \$FFFFFFFF or 0 to \$7F) displacements that will be sign-extended to 32 bits at run time (e.g., the operand of MOVEQ). In the more common case of immediate values which are not sign-extended at runtime, the expected range is 0 to \$FF or \$FFFFFF00 to \$FFFFFFFF. In either case the value inserted in the object module is the low 8 bits of the complete 32-bit value, whether this error is reported or not. This message interrupts the Load Map when it appears. The section and location relative to the beginning of the subsection (i.e., the address that appears on the assembler listing) are given for each occurrence. The module is shown in the preceding line of the Load Map.

(323) **16-bits Value Out of Range at nnnn in module xxxx section yyyy.**

The relocated value of an expression will not fit into a 16-bit field. Loading will continue, but the program may not run properly. You should investigate this warning.

For example, an absolute short instruction refers to a location that is not in the range \$0 through \$7FFF or \$FFFF8000 through \$FFFFFFFF. A PC-plus-16-bit-displacement instruction may refer to a location that is more than +/-32K bytes from the present location.

Often, this error occurs in conjunction with an "Unresolved External" error. The loader assigns the value zero to undefined symbols and then tries to reference address 0.

All expressions are evaluated as unsigned 32-bit values. If a 16-bit field will be sign-extended at run-time, then the value must fall within the range \$0

Loader Error Messages  
(324)–(331)

through \$7FFF or \$FFFF8000 through \$FFFFFFFF. If the field will not be sign-extended, then the value must fall in the range \$0 through \$FFFF or \$FFFF0000 through \$FFFFFFFF.

In any case, the value inserted into the field is the low 16 bits of the value.

(324) **Section Mismatch Between Symbol Def and Ref.**

An XREF from the assembler had a section associated with it which does not match the section of the XDEF with the same name, or does not match the section associated with a previous XREF to the same symbol. Unspecified sections are considered to match any section name. The symbol is treated as undefined. (This message may occur in the case of duplicate XDEFs as well.)

(325) **Illegal HP section name.**

The HP object file contains an illegal section name.

(326) **Cannot open temporary file.**

(327) **Illegal ALIAS command.**

(328) **Illegal command for ALIAS section.**

A section that was ALIASed to another section was mentioned in a loader command. The original section name should not be referenced.

(329) **Multiple initialization of a COMMON section.**

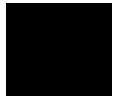
This error occurs when more than one file defines data or instructions (as opposed to just reserving space) in the same COMMON section. Since each file's contribution to a COMMON section will overlap, data from one file may overwrite data from a second file.

(330) **Illegal ALIAS for a COMMON section.**

(331) **Inconsistent IEEE object format.**

The loader has encountered a relocatable module that it cannot properly interpret. Usually, this results from using different versions of assembler and loader programs. A later version of the assembler will produce a relocatable that is rejected by an earlier version of the loader.

- (332)           **Object contains errors.**  
The assembler detected errors when the relocatable module was produced. The module may contain code that will not execute properly.
- (333)           **Source file does not exist.**  
This warning indicates that a source file could not be found in the same place where it was compiled and assembled.
- (334)           **Local symbols in CODE section.**  
This warning can occur if you are linking **.o** files which were assembled with version 1.20 or earlier of the assembler. Re-assemble the files.
- (335)           **Local symbols in DATA section.**  
This warning can occur if you are linking **.o** files which were assembled with version 1.20 or earlier of the assembler. Re-assemble the files.
- (336)           **Local symbols in COMN section.**  
When assembling with the "generate HP format output files" option, more than one relocatable section was mapped to HP section DATA. Local symbols from these extra sections are not written to the "asmb\_sym" assembler symbol file and will not be available for debugging. To eliminate this warning, move the extra sections to a new source module.  
  
This warning can occur if you are linking **.o** files which were assembled with version 1.20 or earlier of the assembler. Re-assemble the files.
- (337)           **Illegal command for incremental linking.**  
Only LOAD commands are allowed during an incremental link.
- (338)           **Duplicate ROM section.**  
More than one INITDATA command was issued.



Loader Error Messages  
**(339)–(347)**

**(339) Section moved to high short section.**

The loader was locating short sections in low base page (\$0000 through \$7FFF). It encountered a short section which would not fit in low base page. It located the section in high base page. The addresses of high base page depend on which microprocessor was specified with the CHIP command:

68008                   \$000F8000 through \$000FFFFFFF.

68000/10/332           \$00FF8000 through \$00FFFFFFF.

68020/30/40           \$FFFF8000 through \$FFFFFFF.

68030                   \$FFFF8000 through \$FFFFFFF.

68040                   \$FFFF8000 through \$FFFFFFF.

This value may be modified by the loader CHIP command.

**(340) Out of virtual memory.**

You have exceeded the host system's limits for process size.

**(341) This command is illegal after LOAD is used.**

**(342) Incompatible incrementally linked object. Recreate the object.**

The linker has read an incrementally linked relocatable file produced by the HP 64870 68000/10/20 linker version 1.20. Because of a defect in the earlier version, the file must be remade before it will be accepted by ld68k. You may redo the incremental link with ld68k version 1.30 or later.

**(343)**

**I/O Error.**

**(345) Duplicate Public From Library Module -- ignored.**

A public symbol was already defined in the library.

**(346) Could Not Construct Full Path Name:**

Check that the objects you are trying to link are on the same host computer.

**(347) Command Ignored:**

- (348) **Module Not Found,**
- (349) **Section Previously Specified Or Non\_existent:**
- (350) **Illegal Multiple Case Specification for *class***  
Each class (PUBLICS, MODULES, SECTIONS) can have only one case specification (CASE, UPPERCASE, LOWERCASE).
- (351) **Write error - disk may be full.**  
An I/O error occurred while writing the output file. The output file will have been removed if this error occurs.
- (352) **Section mismatch between PUBLIC def and Module ref for symbol *xxx***
- (353) **Redefinition of *xxx***  
A symbol defined by the PUBLIC command or a register has been redefined. Definitions can be made INDEX, PUBLIC, or XDEF commands. The value of the symbol is the value specified by the last PUBLIC command.
- (362) **Too Many Errors**  
Any errors found after this message is shown will not be reported.
- (364) **Cannot ABSOLUTE unknown section,**  
This section is not defined in any of the modules loaded by the linker.
- (365) **Cannot ALIGN unknown section,**  
This section is not defined in any of the modules loaded by the linker.
- (366) **Cannot ALIGN absolute section,**  
Absolute sections have a fixed starting address and cannot be aligned.
- (367) **Absolute section cannot have the same name as other sections,**  
Absolute sections cannot be combined with relocatable sections.
- (368) **Combined section exceeds memory space,**
- (372) **Section size shrunk for**  
The default size of a COMMON section was greater than specified by the SECTSIZE command.

Loader Error Messages  
(373)–(380)

- (373)           **24-bit Value Out of Range at**
- The relocated value of an expression will not fit into a 24-bit field. Although loading continues, the program may not run properly.
- (374)           **ORDER command could not be obeyed for section,**
- An impossible section order was specified in the ORDER command. For example,
- ```
ORDER sect1,sect2,sect3
SECT sect2=0
```
- Since sect2 must begin at address 0, sect1 cannot precede sect2.
- (375)           **No modules were loaded**
- No LOAD or LOAD\_SYMBOLS commands were specified. Another error may have prevented the linker from reading the LOAD command. Note that a library will not be loaded if there are no undefined externals.
- (376)           **Invalid modifier, *modifier***
- (377)           **Duplicate section name specified in INITDATA command(s)**
- (379)           **Invalid INITDATA command**
- The INITDATA command is missing operands.
- (380)           **\*\* is no longer a valid comment character in this context**
- Use a semicolon (;) to begin comments in linker command files. (This is a change from earlier versions of the assembler.)

---

# C



---

## Librarian Error Messages

This chapter describes the error messages and warnings that may appear while executing the librarian.

## Librarian Error Messages (100)–(107)

The librarian writes error messages to the current listing device. Some errors are fatal, and some are warnings, depending upon the circumstances of the particular operation. See the "Error Message Formats" appendix for explanations of librarian error severity levels.

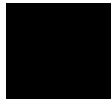
After executing the librarian, you should review the listing to make certain that all commands have been properly processed. A message is written to the listing device each time a library is written into a file.

The errors and messages for the librarian are listed and described below.

- (100)       **Could not close file [filename] to open another file.**
- The librarian attempts to keep as many files open as it can to reduce overhead. If too many files are open it must close one to open a new one.
- (101)       **Unable to open file [filename] in mode [module name].**
- The librarian received an error when trying to open the named file in the named module.
- (102)       **Unable to close file [filename].**
- The librarian received an error when trying to close the named file.
- (103)       **Unable to open input file [filename].**
- The librarian received an error when trying to open the named file.
- (104)       **File [filename] not included.**
- The contents of the named file will not be included in the library.
- (105)       **File not included.**
- The named file has not been included in the library.
- (106)       **File [filename] exists already.**
- This message appears if the CREATE command is used and the library name exists as a file already.
- (107)       **File [filename] does not exist.**
- This message appears if the OPEN command is used and there is no such file.



- (108)           **Library file [libname] not opened.**  
This message appears if the OPEN command is used but the library could not be opened.
- (109)           **Library file [libname] not included.**  
The contents of the named library file are not included in the current library. This message appears if the ADDLIB command is used and the module cannot be included in the current library.
- (200)           **Module [module name] not found.**  
A named module was not found in the target library.
- (201)           **Module [module name] not found.**  
A named module was not found in the target library.
- (202)           **Module [module name] not found in current library.**  
The named module was not found in the library being built. Check the spelling.
- (203)           **Module [module name] already exists in current library.**  
Duplicate module names.
- (204)           **[filename] is a library file.**  
This is an informative message that appears when the librarian was looking for a module file.
- (205)           **[filename] is not a library file.**  
This is an informative message that appears when the librarian was looking for a library file.
- (206)           **Module [module name] is not being included in the library.**  
Self explanatory.
- (207)           **Bad object record.**  
Either the object module has been corrupted or it is not a HP-MRI IEEE 695 relocatable object file.



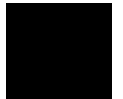
Librarian Error Messages  
(208)–(258)

- (208)           **Bad library header record.**  
The library may have been corrupted.
- (209)           **Duplicate symbol [filename].**  
Two different modules have the same public definition symbol. The Librarian is always case sensitive with symbols.
- (210)           **Bad object record in file [filename].**  
The library or module file may have been corrupted.
- (250)           **Out of memory.**  
The librarian could not allocate any more memory from the system.
- (251)           **Failed writing library.**  
This message is always preceded by the precise reason for the failure.
- (252)           **Fseek or ftell error.**  
It is possible that one of the object files used to build the library has been corrupted.
-  (253)           **Library [libname] not written.**  
This message is always preceded by the precise reason for the failure.
- (254)           **Failed writing module [module name] to file [filename].**
- (255)           **Replacement not done.**  
The librarian was unable to perform the REPLACE as specified.
- (256)           **Extraction Failed.**  
The librarian was unable to perform the EXTRACT as specified.
- (257)           **Illegal command.**  
Retype the command or argument. This message could also mean that the user attempted to start the command sequence with the ADDMOD command.
- (258)           **Abrupt ending of comment.**  
There was a new line before the second quote.

(259)

**Quote not terminated.**

There was a new line before the second quote.

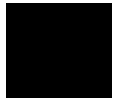


Librarian Error Messages  
**(259)–(259)**



---

# D



---

## Error Message Formats

This chapter explains the difference between warnings, errors, and fatal errors.

## Error Classes

There are three classes of errors that may occur during assembler, linker/loader, or librarian execution: warnings, errors, and fatal errors.

### Warnings

Warnings announce something that *might* be a problem in the output file. For example, the loader warns of a section mismatch between the definition and reference of a symbol. This may or may not indicate a problem with the program.

After a warning, the output files are written normally.

After a warning ar68k, as68k, and ld68k return a return code indicating "success" so that command files and "make" operations continue normally.

### Errors

Errors announce something that IS wrong in the output file. For example, an unresolved external symbol will cause a loader error. A reference to an unresolved symbol will cause problems at run-time.

After an error, the output files are written normally. The output files are complete and may be useful in subsequent operations.

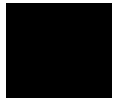
After an error ar68k, as68k, and ld68k return a return code indicating "error" so that command files and "make" operations stop.

## **Fatal Errors**

A fatal error announces a condition that causes processing to be discontinued. For example, the linker/loader produces a fatal error when one of its input modules is not a valid IEEE relocatable file.

After a fatal error, the output files are incomplete and corrupt. They are not useful for subsequent operations.

After an error ar68k, as68k, and ld68k return a return code indicating "error" so that command files and "make" operations stop.



## Interactive and Non-Interactive Conditions

Some conditions produce either warnings or errors, depending on whether the tool is run in interactive or batch mode. In interactive mode, a particular condition causes a warning because the user has a chance to reissue the command correctly. In batch mode, the same condition causes an error.

For example (on the HP-UX operating system), suppose the file **tt2.o** does not exist. If we invoked the librarian in batch mode as follows:

```
$ ar68k -a "tt2.o" lib.a
```

We would see an error.

```
< ar68k >
          (101) unable to open file tt2.o.
ERROR:   (104) file tt2.o not included.
          (253) Library lib.a not written.

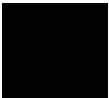
Warnings = 0
Errors   = 1
```

In interactive mode, if we typed the following command:

```
ar68k> addmod tt2.o
```

We would see a warning.

```
          (101) unable to open file tt2.o.
WARNING: (104) file tt2.o not included.
```





---

# E

---

## Converting to HP B3641 Assembly Language



This appendix describes how you can convert source files written for the HP 64845 assembler so that they will work with the 68000 Family Assembler/Linker/Librarian.

## **Converting HP 64845 Assembly Language Programs**

This appendix documents the changes that must be made to source files written for the HP 64845 assembler so that they can be assembled with the HP B3641 assembler. Not everything that appears in the HP 64845 format source files can be translated into something which the HP B3641 assembler will recognize, but a good portion can.

Source file conversion utilities may be supplied with the assembler as “contributed software.” These utilities, if supplied, **will not be supported** by Hewlett-Packard.

---

### **Note**

Some of the source file conversions described in this appendix will allow instructions to be assembled with no errors on the HP B3641 assembler. However, the relocatable object code generated may not always be the same. Identical instructions may cause different code to be generated due to the method in which the assembler chooses addressing modes or optimizes instructions. For example, given a source file line of "MOVE.L # 1,D0", the HP 64845 assembler will generate code for a MOVE instruction with two words of extension while the HP B3641 assembler will generate a MOVEQ instruction with zero words of extension.

---

## Converting to HP B3641 Assembly Language Converting HP 64845 Assembly Language Programs

Labels, assembly language instructions, numeric terms, and comments will not have to be changed. Areas which require changes are listed below.

- **Chip Directives.** You are required to tell the HP 64845 assembler to generate instructions for a certain microprocessor by including a string in the first line & column of the source file. For example:

```
"68000"  
^68010^  
'68008'
```

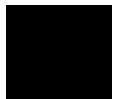
Depending on what your target processor is, you either replace the chip string with the appropriate HP B3641 assembler CHIP directive, or remove the chip string from the source file altogether. In the absence of a CHIP directive (or equivalent command line option), the B3641 defaults to the 68000 processor. CHIP directives equivalent to the preceding chip strings are as follows:

```
CHIP 68000  
CHIP 68010  
CHIP 68008
```

- **Flags.** The HP B3641 assembler has flags that affect its behavior (see the OPT assembler directive). For the HP B3641 assembler to operate in a manner that is most like the HP 64845 assembler, you should always include the following directives in programs to be assembled by the HP B3641 assembler.

```
OPT NOABSPCADD ; Absolute expressions in PC-relative operands  
                ; are treated as displacements.  
OPT NOPCR      ; Do not optimize absolute operands to be PC-relative.
```

- **Pseudo-Ops.** Some pseudo-ops used in HP 64845 source files have comparable directives in the HP B3641 product. See "Converting HP 64845 Pseudo-Ops" later in this appendix for more information.
- **Operand Symbols and Delimiters.** Various operand symbols and delimiters will have to be modified; for example, the HP 64845 uses brackets where 68000/10 syntax specifies parentheses, the symbol for "current assembly location counter" is different, string delimiters are different, and logical operators have different forms. See "Converting HP 64845 Operands" later in this appendix for more information.



Converting to HP B3641 Assembly Language  
**Converting HP 64845 Assembly Language Programs**

- **Character Strings.** Character strings are packed differently in words or longwords. See “Converting Character Constants” later in this appendix for more information.
- **Macros.** Macros are similar; however, there are some fundamental differences between HP 64845 macros and HP B3641 macros. See “Converting HP 64845 Macros” later in this appendix for more information.
- **Miscellaneous.** The HP 64845 assembler sometimes allows white space where the HP B3641 assembler does not. See “Converting HP 64845—Miscellaneous” for more information about this and other miscellaneous conversions.

In addition to issues surrounding conversion of HP 64845 assembler files for use with the HP B3641 assembler, there are issues with using incrementally linked or library files created with earlier versions of the HP 64870 assembler. Refer to “Compatibility with older HP 64870 Files” later in this appendix for further information.

## Converting HP 64845 Pseudo-Ops

Listed below are the pseudo-ops allowed in HP 64845 source files and their counterparts (if any) in the HP B3641 assembler.

HP 64845

HP B3641

**ABSOLUTE\_LONG** No substitute. The OPT FRL directive in the HP B3641 assembler is used to force absolute long addressing in forward references only. The **ABS\_LONG** pseudo-op will force the absolute long address mode in forward and backward references. To force the absolute long address mode, use the .L extension on individual operands (e.g., "<exp>.L").

**ABSOLUTE\_SHORT** No substitute. The OPT FRS directive in the HP B3641 assembler is used to force absolute short addressing in forward references only. The **ABS\_SHORT** pseudo-op will force the absolute short address mode in forward and backward references. To force the absolute short address mode, use the .W extension on individual operands (e.g., "<exp>.W").

**ASCII/ASC**

**DC.B**

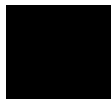
**A5\_REL\_ON**  
**A5\_REL\_OFF**

No substitute. The linker/loader INDEX command provides for A2-A5 relative addressing. If the **A5\_REL\_ON** pseudo-op is used, be sure to use the linker/loader INDEX command and specify the run-time value of "An" as the value you would assign to A5 when answering the HP 64845 "PROG,DATA,COMN,A5?" linker question.

**BINARY/BIN**

**DC**

The operand must be specified as a binary number by adding a "%" prefix or a "B" suffix. (Warnings will be generated if the operand of the DC directive must be truncated to fit into 16-bits. The **BIN** pseudo-op will



Converting to HP B3641 Assembly Language  
**Converting HP 64845 Pseudo-Ops**

also truncate to word lengths, but no warning is generated.)

**COMN**                    **COMMON COMN**

**DATA**                    **SECTION DATA,,D**

**DECIMAL/DEC**           **DC**

(Warnings will be generated if the operand of the DC directive must be truncated to fit into 16-bits. The DECIMAL pseudo-op will also truncate to word lengths, but no warning is generated.)

**END**                      **END**

**EQU**                      **EQU**

**EVEN**                    **ALIGN 2**

**EXPAND**                 **OPT C,I,M,ML,MC**

**EXTERNAL**               **XREF**

**GLOBAL/GLB**            **XDEF**

**HEX**                      **DC**

The operand must be specified as a hexadecimal number by adding a "\$" prefix or a "H" suffix. (Warnings will be generated if the operand of the DC directive must be truncated to fit into 16-bits. The HEX pseudo-op will also truncate to word lengths, but no warning is generated.)

**IF/ELSE/  
ENDIF/IFEND**            **IFNE/ELSEC/ENDC**

**INCLUDE**                 **INCLUDE**

**LIST**

**LIST, OPT S**

**MASK**

No substitute. It is possible to duplicate this operation by ANDing and ORing each character in the ASCII pseudo instruction's operand with values defined in SET directives.

```
* HP 64845 Instructions:
*-----
      MASK      77H,101B
      ASCII     'abcd'
      MASK      0A5H
      ASCII     'ef'

* HP B3641 Equivalent:
*-----
AND_VAL SET      77H
OR_VAL  SET      101B
DC.B    'a'&AND_VAL!OR_VAL
DC.B    'b'&AND_VAL!OR_VAL
DC.B    'c'&AND_VAL!OR_VAL
DC.B    'd'&AND_VAL!OR_VAL
AND_VAL SET      0A5H
DC.B    'e'&AND_VAL!OR_VAL
DC.B    'f'&AND_VAL!OR_VAL
```

Remember, MASK only affects strings defined with the ASC/ASCII pseudo instruction.

**NAME**

No substitute.

**NOLIST**

**NOLIST, OPT -S**

**NOWARN**

**OPT W**

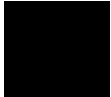
**OCT/OCTAL**

**DC**

The operand must be specified as an octal number by adding a "@" prefix or "O" or "Q" suffixes. (Warnings will be generated if the operand of the DC directive must be truncated to fit into 16-bits. The OCT pseudo-op will also truncate to word lengths, but no warning is generated.)

**ORG**

**ORG**



Converting to HP B3641 Assembly Language  
**Converting HP 64845 Pseudo-Ops**

**PROG**                    **SECTION PROG,,P**

**REAL**                    **DC.S, DC.D**

In the HP 64845 assembler, short reals are generated by using "E" to specify the power of ten (e.g., 1.0E2) and long reals are generated by using "L" to specify the power of ten (e.g., 1.0L2). With the HP B3641 assembler, always use "E" to specify the exponent and use "DC.S" to generate short reals or "DC.D" to generate long reals.

**REPT**                    **REPT and ENDR**

The HP 64845 assembler allows you to repeat one statement. The HP B3641 assembler allows you to repeat a number of statements (the statements between REPT and ENDR); therefore, you must add the ENDR directive after the statement which is to be repeated.

**RORG**                    **OPT NOPCR,NOPCS**

The RORG directive is not exactly equal to OPT NOPCR,NOPCS. The RORG directive affects "absolute to relocatable" references and "relocatable to absolute" references. No HP B3641 flag does this.

**SET**                    **SET**

**SKIP**                    **PAGE**

**SPC**                    **SPC**

**TITLE**                    **TTL**

**WARN**                    **OPT -W**



## Converting HP 64845 Operands

Arithmetic operators and numeric terms in HP 64845 operands do not have to be changed before they are assembled with the HP B3641 assembler.

The HP 64845 assembler allowed spaces after commas in some operand lists (e.g., lists of symbols in the GLOBAL and EXTERNAL pseudo-op operands). The HP B3641 assembler will not allow spaces in operand lists. Any time a space appears in an operand, the remainder of the line is interpreted as a comment.

Other parts of the operand fields which must be changed are shown below.

HP 64845

HP B3641

[ ]

( )

The HP 64845 assembler syntax requires brackets when using the indirect address mode operands. These brackets should be changed to parentheses before assembling with the HP B3641 assembler.

\$

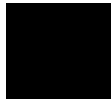
\*

Current assembly location counter symbol.

^ " ' ,

' ^

The HP 64845 assembler allowed three types of string delimiters. In the HP B3641 source file, only the single quote (') and the caret (^) can be used as string delimiters. When using the single quote character as a string delimiter and you wish to include a single quote as part of the string, use two adjacent single quotes. The same is true for the caret character.



Converting to HP B3641 Assembly Language  
**Converting HP 64845 Operands**

### Converting Character Constants

Be careful when using character constants as word or longword operands. The HP 64845 assembler right justifies character constants. The HP B3641 assembler left justifies character constants on word or longword boundaries. For example:

```
MOVE.L  #'A',D0          ; HP 64845 moves $00000041.  
                          ; HP B3641 moves $00004100.
```

### Converting Logical Operators

Different symbols are used for logical operators in the HP 64845 assembler. The HP B3641 equivalents are shown below.

| HP 64845    | HP B3641                              |
|-------------|---------------------------------------|
| <b>.AN.</b> | <b>&amp;</b><br>Logical AND.          |
| <b>.NT.</b> | <b>"</b><br>Logical one's complement. |
| <b>.OR.</b> | <b>!</b><br>Logical OR.               |
| <b>.SL.</b> | <b>&lt;&lt;</b><br>Shift left.        |
| <b>.SR.</b> | <b>&gt;&gt;</b><br>Shift right.       |

## Converting HP 64845 Macros

There are some fundamental differences between macros in the HP 64845 assembler and macros in the HP B3641 assembler. The HP 64845 assembler provides greater flexibility with its conditional macro assembly instructions, and the capability offered by these conditional instructions cannot be completely duplicated by the HP B3641 assembler. However, other parts of HP 64845 macros are similar to HP macros.

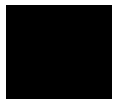
### Macro Headings

Macro headings are the same in both assemblers with one exception: macro parameters must begin with the ampersand (&) character in the HP 64845 assembler. The ampersand is a special character in the HP B3641 assembler and will cause errors if used in macro parameters. The solution to this problem is: 1) remove the ampersand character in the macro definition line, and 2) precede the parameter with "&&" in the macro body.

### Unique Label Generation

In the HP 64845 assembler, unique local labels are created whenever a macro is called by using four ampersand characters (&&&&) in macro definition labels. The HP B3641 assembler uses the "\@" characters to accomplish the same thing. When converting HP 64845 macros, replace every occurrence of "&&&&" with "\@" before assembling with the HP B3641 assembler.

Be aware that the unique local symbols generated are not the same. For example, suppose you specify the "LABEL&&&&" local symbol in a HP 64845 macro definition. The counterpart in the HP B3641 would be "LABEL\@". On the first macro call in the HP 64845 assembler, the symbol created would be "LABEL0001". With the HP B3641 assembler, the first macro call would create the symbol "?0001".



## Conditional Assembly Within Macros

The HP 64845 assembler provided for conditional assembly within macros with four conditional instructions:

```
.SET  
.IF  
.GOTO  
.NOP
```

### The ".SET" Instruction

The ".SET" conditional instruction in HP 64845 macros can be replaced with the HP B3641 "SET" directive.

### The ".IF" Conditional Branch Instruction

In the HP 64845 assembler, the ".IF" instruction is a conditional branch instruction that uses six relational operators:

```
.EQ.  
.NE.  
.LT.  
.GT.  
.LE.  
.GE.
```

The ".IF" conditional branch instruction has the following format:

| <u>Label</u> | <u>Operation</u> | <u>Operand</u>                       | <u>Comment</u> |
|--------------|------------------|--------------------------------------|----------------|
|              | .IF              | <exp> .<relational operator> . <exp> | label          |

If the value of the comparison is true, the HP 64845 assembler goes to the "label" in the macro definition and continues to process the macro definition instructions from that statement.

While you can set up (in the HP B3641 assembler) a macro definition that contains a conditional macro call to itself, there is no way for a macro to call parts of itself. The ".IF" instruction cannot be duplicated.

### **The ".GOTO" Unconditional Branch Instruction**

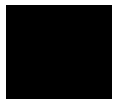
The ".GOTO" unconditional branch instruction cannot be duplicated because of the same reasons listed above for the ".IF" conditional branch instruction.

### **The ".NOP" Instruction**

The ".NOP" instruction is a no-operation instruction, and the effect is the same as if the assembler were to begin processing the statement immediately following this instruction. The ".NOP" instruction can be used with the ".IF" and ".GOTO" conditional instructions to exit the macro conditionally. While the HP B3641 assembler does not provide anything similar to the ".NOP" macro instruction, it does provide the MEXIT macro instruction to exit a macro.

### **Indexing Parameters**

The HP 64845 assembler provides a way to index parameters in a macro parameter list by using two ampersands and a macro local symbol (e.g., &&SYMB). The macro local symbol SYMB is usually set to equal a number, or possibly a macro parameter, with the ".SET" instruction. The HP B3641 assembler also provides parameter indexing with the "\n" macro operator (where n = the number of the parameter). Symbols set equal to numbers are not allowed with the "\n" operator. To convert "&&SYMB" to "\n", you must backtrack to find the number that SYMB equals and substitute that number for "n".



## Converting HP 64845— Miscellaneous

### White Space

The HP 64845 assembler sometimes allows white space in lists of operands. This white space will cause errors when assembled with the HP B3641 assembler. For example,

```
EXTERNAL      LAB1 ,  LAB2
```

should be rewritten as:

```
XREF      LAB1 , LAB2
```

### White Space in Macro Parameters

In the HP B3641 assembly language, white space delimits an actual macro parameter, even inside quoted strings. Therefore, strings containing white space should be surrounded by angle brackets (< , >) as shown in the following example.

```
M1      MACRO  P1  
        .  
        ENDM  
M1      <"TWO WORDS" >
```

## Compatibility with older HP 64870 and HP 64874 Files

If you have been using an older revision of the HP 64870/B1464 68000/10/20/332 Assembler Linker Librarian or the HP 64874 68030/40 Assembler Linker Librarian, you can still use your old relocatable, library, and source files with the HP B3641. When using older files, there are three areas for compatibility that must be considered:

- Relocatable and Library Files
- Assembler Source Files
- Linker Command Files

---

### Note

In the following text, the CHIP directive is mentioned several times. Anywhere the CHIP directive can be used, the OPT P= directive can also be used to specify the target processor.

---

### Relocatable and Library Files

Relocatable files produced by older versions of the the HP 64870 and HP 64874 and library files produced by the most recent version of the HP 64870 and HP 64874 Librarian are accepted, unchanged, as input files by the HP B3641.

The reverse is not always true. Relocatable and library files produced by the HP B3641 Assembler/Linker/Librarian will **not** be accepted by older versions of the HP 64870 or HP 64874 assemblers if the CHIP directive used when these files were created is not one that is accepted by the HP 64870 or HP 64874. (For example, a CHIP 68030 directive used with the HP 64870, or a CHIP 68020 directive used with the HP 64874.) One of the two following error messages will occur:

```
FATAL ERROR: (300) Bad IEEE Object Record Module: Part: Header  
Position: 0
```

or

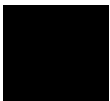
```
ERROR: (314) Chip inconsistent
```

Converting to HP B3641 Assembly Language  
**Compatibility with older HP 64870 and HP 64874 Files**

In addition to the CHIP directive problem, changes in the output module format (OMF) will prevent the HP B3641 linker from linking output from versions of the HP 64870 assembler that support the old OMF.

**Assembly Source Files**

Assembly files used with the HP 64874 Assembler/Linker/Librarian do not require CHIP directives to identify the target processor, although if they are present (and correct), they do not cause an error. If you use these same files with the HP B3641, and wish to target the 68030 or 68040 processor, you must include explicit CHIP directives in the files or specify the target processor on the command line when you invoke the assembler.





---

**F**



---

## **About this Version**

How this version of the assembler differs from previous versions.

## Version 2.01

### PC Platform Support

The assembler is now available for personal computers running MS-DOS.

### Re-organized manual

The *User's Guide* and *Reference* manuals have been combined and the chapters have been re-organized a bit.

---

## Version 2.00

*Note: These changes may require you to change your linker command files.*

### Combined products

The HP B3641 68000 Family Assembler/Linker/Librarian combines the HP 64870 and HP 64874 products into a single assembler that supports all Motorola 68000 family microprocessors.

### New features: as68k

- Byte relocation is now supported.
- The operators \* and / are now allowed in relocatable expressions.
- The assembler now warns when extra operands are detected for assembler directives.
- Passing a string that contains a space as a macro parameter no longer causes improper macro expansion.



- A label on a odd address and on a line by itself now gets the odd address, rather than the address of the next even aligned instruction.
- Sections may now have an alignment attribute of 1, in addition to alignments of 2 and 4.
- Positional parameters and local macro labels *cannot* be placed in the label part of an ENDM (macro terminator) directive. The same effect can be achieved by placing the label on the line prior to the ENDM directive.
- The EQU directive now supports simple forward references (a symbol with no operands) and allows constant offsets to be added or subtracted from external symbols.
- A coprocessor id field number of 7 is now accepted correctly as an operand by the FOPT directive.
- The IFDEF and IFNDEF directives have been added. These directives control conditional assembly based on whether a symbol is defined or not defined.
- The syntax of the ORG directive has been extended to allow absolute sections to be named.
- The CHIP directive now accepts additional chip types. The valid chip types accepted and corresponding instruction sets allowed are as follows:

| <b>Input Processor String</b> | <b>Instruction Set Allowed</b> |
|-------------------------------|--------------------------------|
| 68000                         | 68000                          |
| 68EC000                       | 68000                          |
| 68HC000                       | 68000                          |
| 68HC001                       | 68000                          |
| 68008                         | 68000                          |
| 68010                         | 68010                          |
| 68302                         | CPU32                          |
| 68330                         | CPU32                          |
| 68331                         | CPU32                          |



|         |         |
|---------|---------|
| 68332   | CPU32   |
| 68333   | CPU32   |
| 68340   | CPU32   |
| CPU32   | CPU32   |
| 68020   | 68020   |
| 68EC020 | 68020   |
| 68030   | 68030   |
| 68EC030 | 68EC030 |
| 68040   | 68040   |
| 68EC040 | 68EC040 |

The CHIP directive no longer supports an absolute expression. Processor types are processed as strings.

- The behavior of the XDEF and XREF directives has been changed. An XDEF will override a previous XREF for any symbol that has not already been defined.

### **New features: ld68k**

- The symbol table manager has been enhanced. This results in as much as a 30 percent performance increase on large links.
- Static functions and static variables noew appear in the Local Symbol Table section of the map file.
- The comment character in linker command files has been changed from an asterisk (\*) to a semicolon (;). The asterisk can still be used as a comment character only if it is placed in column 1. Usage in other than column 1 causes a collision with the use of '\*' with the MERGE command.
- The command continuation character in linker command files is now the pound sign (#) instead of the plus sign (+).
- The escape character in linker command files is now a single quote (') instead of the dash (-).

- The linker can now link an unlimited number of modules. In previous versions, exceeding the limit of 500 modules could cause a core dump.
- The linker now generates an entry for the `\0` section in the `MODULE SUMMARY` of the map file.
- Section names in the `SECTION SUMMARY` of the map file are no longer limited to 8 characters.
- The linker will now generate an error if no modules are loaded.
- The `LIST` and `NLIST` commands will not be supported in future releases of the linker.
- The `LISTMAP` command now has an option to set the page length for the map file listing.
- The new `LOAD_SYMBOLS` command instructs the linker to allocate space for a module, but to only load symbols and debug information from the module.
- The `ORDER` and `SORDER` commands now accept a section type argument.
- The `PUBLIC` linker command now allows `PUBLIC sym1= sym2+ offset` syntax.
- The new `SECTSIZE` command can be used to change section sizes at link time.
- The `START` command will now accept a symbol or a value as an argument.



Chapter F: About this Version  
**Version 2.00**



---

# Index

- "
  - " in linker command files **201**
- \$
  - hexadecimal constant prefix **201**
- \*
  - in linker command files **203**
  - See also* location counter
  - wild card in linker command files **238**
- \_, in HP format files **192**
- A**
  - A2-A5 relative addressing **64 – 71**
  - A2-A5 relative addressing, example **67**
  - ABSOLUTE loader command **205 – 206**
  - absolute long mode **46**
  - absolute section **186**
  - absolute short mode **46**
  - absolute vs. relocatable symbols **82**
  - ADDLIB librarian command **273**
  - ADDMOD librarian command **274**
  - address format in assembler listing **19**
  - address lines **188**
  - address modes **38 – 49**
    - absolute short **187**
    - and the 68881 **48**
    - user control of **62 – 63**
  - address register indirect modes **43**
    - with 8-bit displ. & index (68000 model) **44**
    - with 8-bit displ. & index (68020 model) **44**
    - with base displ. & index (68020 model) **44**
    - with displacement **44**
    - with postincrement **43**
    - with predecrement **44**
  - address register indirect with displacement addressing modes **64**
  - address registers A2-A5 **224 – 225**
  - address rounding **191**

## Index

- addresses
  - even and odd **191**
  - odd locations **43**
- addressing modes
  - 68000 model **40**
  - 68020 model **41**
  - 68332 model **42**
- addressing modes, operand syntax and **52**
- advantages of A2-A5 relative addressing **65**
- ALIAS loader command **207**
- ALIGN assembler directive **90**
- ALIGN loader commands **208**
- alignment (section) attributes **76**
- alignment, section **188**
- ALIGNMOD loader commands **208**
- arguments
  - linker command **201**
- as68k features **2**
- ASCII vs. EBCDIC character strings **14**
- assembler
  - character set **8**
  - constants **12 – 15**
  - directives **85 – 156**
  - error messages **289 – 298**
  - introduction **1 – 20**
  - listing format description **18**
  - statements **3 – 4**
  - structured control directives **170**
  - symbols **9 – 11**
- assembler listing
  - hint for debugging addresses **191**
- assembler syntax
  - rules **50**
- assembly language
  - converting HP 64845 source files **320 – 322**
  - instructions **23**
  - location counter symbol **11**
- assembly program counter
  - See* location counter(\*)
- asterisk, librarian command character **264**



- attributes (section)
  - common vs. noncommon **75**
  - section type **77**
  - short vs. long **76**
- attributes, section **187**
- B** base address **189**
- BASE loader command **190, 209**
- blanks, librarian command file **265**
- BREAK directive (loop exit) **179**
- brief format library listing example, description **269**
- C** C flag in assembler listing **19**
- CASE loader commands **210 – 211**
- case sensitivity
  - in linker **201**
- character constants **14**
- character constants, HP 64845 and HP B3641 **328**
- character set (assembler) **8**
- CHIP assembler directive **91 – 92**
- CHIP loader commands **212 – 213**
- CLEAR librarian command **275**
- COMLINE assembler directive **93**
- comma, librarian command character **264**
- command format, linker **201**
- command syntax, librarian **264 – 265**
- comment field **4**
- comment statement **6**
- comments
  - librarian command file **265**
  - linker command file (;) **203**
- comments, loader **203**
- COMMON assembler directive **94 – 95**
- COMMON loader command **214**
- common section **187**
- common vs. noncommon section attributes **75**
- complex relocatable expressions **83 – 84**
- conditional execution, IF...THEN...ELSE...ENDI **175 – 176**

## Index

- constants **12 – 15**
  - character **14**
  - integer **12**
  - linker command format **201**
- contents, section **188**
- continuation, loader **204**
- CONTROL register **37**
- converting HP 64845 source files
  - conditional assembly within macros **330**
  - logical operators **328**
  - macro headings **329**
  - macro indexing parameters **331**
  - macros **329 – 331**
  - operands **327 – 328**
  - pseudo-ops **323 – 326**
- CPAGE loader command **215**
- CREATE librarian command **276**
- cross reference table format **20**
- D**
  - data
    - in ROM **226 – 228**
    - sharing between sections **187**
  - DC assembler directive **96 – 98**
  - DCB assembler directive **99 – 100**
  - DELETE librarian command **277**
  - directives
    - list of **86**
    - ALIGN **90**
    - assembler **85 – 156**
    - BREAK (structured control) **179**
    - CHIP **91 – 92**
    - COMLINE **93**
    - COMMON **94 – 95**
    - DC **96 – 98**
    - DCB **99 – 100**
    - DS **101 – 102**
    - ELSE (structured syntax) **175 – 176**
    - ELSEC **103**
    - END **104**
    - ENDC **105**
    - ENDF **173 – 174**

directives (continued)

- ENDI 175 – 176**
- ENDM 161**
- ENDR 106**
- ENDW (structured syntax) 178**
- EQU 107 – 108**
- FAIL 109**
- FEQU 110 – 111**
- FILE 112**
- FOPT 113**
- FOR 173 – 174**
- FORMAT 114**
- IDNT 115**
- IF (structured syntax) 175 – 176**
- IF GT 116**
- IFC 117 – 118**
- IFDEF 119**
- IFEQ 116**
- IFGE 116**
- IFLE 116**
- IFLT 116**
- IFNC 117 – 118**
- IFNDEF 119**
- IFNE 116**
- INCLUDE 120**
- INTFILE 121**
- IRP 122**
- IRPC 123**
- LIST 124**
- LLEN 125**
- LOCAL 165 – 166**
- MACRO 159**
- MASK2 126**
- MEXIT 167**
- NAME 127**
- NEXT (structured control) 179**
- NOFORMAT 114**
- NOLIST 128**
- NOOBJ 129**
- NOPAGE 130**

- directives (continued)
  - OFFSET **131 – 132**
  - OPT **133 – 138**
  - ORG **139 – 140**
  - PAGE **141**
  - PLEN **142**
  - REG **143**
  - REPEAT (structured syntax) **177**
  - REPT **144**
  - RESTORE **145**
  - SAVE **146**
  - SECT/SECTION **147 – 148**
  - SET **149**
  - SPC **150**
  - structured control **170**
  - TTL **151**
  - UNTIL (structured syntax) **177**
  - WHILE (structured syntax) **178**
  - XCOM **152**
  - XDEF **153**
  - XREF **154 – 156**
- DIRECTORY librarian command **278 – 279**
- DS assembler directive **101 – 102**
- dynamically allocated data areas, A2-A5 relative addressing **67**
- E**
  - E flag in assembler listing **19**
  - EBCDIC vs. ASCII character strings **14**
  - ELSE directive (structured syntax) **175 – 176**
  - ELSEC assembler directive **103**
  - END assembler directive **104, 116**
  - END librarian command **280**
  - END loader command **217**
  - ENDC assembler directive **105**
  - ENDF directive (structured syntax) **173 – 174**
  - ENDI directive **175 – 176**
  - ENDM directive **161**
  - ENDR assembler directive **106**
  - ENDW directive (structured syntax) **178**
  - EQU assembler directive **107 – 108**
  - ERROR loader command **218**

- error messages
    - assembler **289 – 298**
    - classes **316 – 317**
    - formats **315 – 318**
    - interactive vs. non-interactive **318**
    - librarian **266, 309 – 314**
    - loader **299 – 308**
  - even addresses **191**
  - example assembly statements **5 – 6**
  - example library listing description **269**
  - EXIT librarian command **280**
  - EXIT loader command **219**
  - expressions **16 – 17**
    - relocatable **83 – 84**
    - structured control **171 – 172**
  - EXTERN loader commands **220**
  - external symbols **81**
  - EXTRACT librarian command **281**
- F**
- FAIL assembler directive **109**
  - features of as68k **2**
  - FEQU assembler directive **110 – 111**
  - FILE assembler directive **112**
  - floating-point constants **13**
  - floating-point coprocessor
    - registers **37**
  - floating-point coprocessor (68881) and address modes **48**
  - FOPT assembler directive **113**
  - FOR directive **173 – 174**
  - FOR...ENDF loop **173 – 174**
  - FORMAT assembler directives **114**
  - FORMAT loader command **221**
  - format of assembler statements **3 – 4**
  - formats for error messages **315 – 318**
  - forward defined symbols, code generation for **61**
  - FP data registers **37**
  - FPCR register **37**
  - FPIAR register **37**
  - FPSR register **37**
  - FULLDIR librarian command **282**
  - function codes, not supported **190**

- H**
  - HELP librarian command **283 – 284**
  - hexadecimal constants in linker commands **201**
  - how a librarian works **258 – 263**
  - how code is generated for forward defined symbols **61**
  - hp format absolute files, generating **192**
  - HP section type **189**
  - HP Section type attribute **78**
  
- I**
  - IADDR register **37**
  - IDNT assembler directive **115**
  - IF directive
    - BREAK not allowed **179**
  - IF directive (structured syntax) **175 – 176**
  - IF...THEN...ELSE...ENDI conditional execution **175 – 176**
  - IFC assembler directive **117 – 118**
  - IFDEF assembler directive **119**
  - IFEQ assembler directive **116**
  - IFGE assembler directive **116**
  - IFGT assembler directive **116**
  - IFLE assembler directive **116**
  - IFLT assembler directive **116**
  - IFNC assembler directive **117 – 118**
  - IFNDEF assembler directive **119**
  - IFNE assembler directive **116**
  - immediate mode **48**
  - INCLUDE assembler directive **120**
  - INCLUDE loader commands **222 – 223**
  - incremental linking **191**
  - INDEX loader command **224 – 225**
    - A2-A5 relative addressing **65**
  - indirect addr. modes, absolute vs. relocatable expressions **64**
  - initcopy routine **227**
  - INITDATA loader command **226 – 228**
  - INITDATA section **226**
  - initdata() routine **226**
  - initializing data **226 – 228**
  - instruction operands **34**
  - instructions (assembly language) **23**
  - integer constants **12**
  - INTFILE assembler directive **121**
  - IRP assembler directive **122**
  - IRPC assembler directive **123**

- K** keywords **170**
- L** label field **4**
  - librarian
    - command characters **264**
    - definition **258**
    - example listing **268**
    - features **257**
    - introduction **255 – 270**
    - listing format description **266**
    - messages and error concepts **316**
    - operation **258**
    - overview **258 – 263**
  - librarian commands **271 – 288**
    - ADDLIB **273**
    - ADDMOD **274**
    - CLEAR **275**
    - CREATE **276**
    - DELETE **277**
    - DIRECTORY **278 – 279**
    - END **280**
    - EXIT **280**
    - EXTRACT **281**
    - FULLDIR **282**
    - HELP **283 – 284**
    - LIST **282**
    - OPEN **285**
    - QUIT **280**
    - REPLACE **286**
    - SAVE **287 – 288**
  - librarian error messages **309 – 314**
  - library listing, brief format example **269 – 270**
  - library modules **190, 220**
  - linker
    - See also* loader
  - linker/loader
    - commands **197 – 254**
    - features **185**
    - introduction **183 – 196**
  - linking **81**
  - linking loader
    - See* loader

## Index

- LIST assembler directive **124**
- LIST librarian command **282**
- LIST loader commands **230 – 232**
- LISTABS loader commands **233**
- listing of structured directives **182**
- listing format
  - assembler **18**
- listings, loader **193 – 196**
- LISTMAP loader commands **234**
- LLEN assembler directive **125**
- load address **189**
- LOAD loader command **235 – 236**
- LOAD\_SYMBOLS loader command **237**
- loader
  - error messages **299 – 308**
  - functions of **186**
  - listing format description **193 – 196**
  - operation **185**
- loader commands **216**
  - ABSOLUTE **205 – 206**
  - ALIAS **207**
  - ALIGN **208**
  - ALIGNMOD **208**
  - BASE **209**
  - CASE **210 – 211**
  - CHIP **212 – 213**
  - comment **203**
  - COMMON **214**
  - continuation **204**
  - CPAGE **215**
  - END **217**
  - ERROR **218**
  - EXIT **219**
  - EXTERN **220**
  - FORMAT **221**
  - INCLUDE **222 – 223**
  - INDEX **224 – 225**
  - INITDATA **226 – 228**
  - INTFILE **229**
  - LIST **230 – 232**



- loader commands (continued)
  - LISTABS 233
  - LISTMAP 234
  - LOAD 235 – 236
  - LOAD\_SYMBOLS 237
  - MERGE 238 – 239
  - NAME 240
  - NLIST 241 – 242
  - NOERROR 218
  - NOPAGE 243
  - ORDER 244 – 246
  - PAGE 247
  - PUBLIC 248 – 249
  - RESADD/RESMEM 250 – 251
  - SECT 252
  - SECTSIZE 253
  - SORDER 244 – 246
  - START 254
  - WARN 218
- loader commands, summary 198
- LOCAL directive 165 – 166
- location counter (\*) 11, 75, 82
- long section 188
- loop exit--BREAK directive 179
- M**
  - macro body 160
  - macro call 162 – 164
  - MACRO directive 159
  - macro heading 159
  - macro parameter count 168
  - macro statement 6
  - macro terminator 161
  - macros 157 – 168
    - names 296
  - macros, MEXIT 167
  - MASK2 assembler directive 126
  - memory allocation, order of 189
  - memory indirect post-indexed mode (68020 model) 45
  - memory indirect pre-indexed mode (68020 model) 45
  - memory space assignment 189 – 190
  - MERGE loader command 238 – 239
  - message severity 218

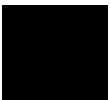
- MEXIT directive **167**
- MMUSR register **35**
- modes (address), user control of **62 – 63**
- module names **265**
- N**
  - NAME assembler directive **127**
  - NAME loader command **240**
  - NARG assembler reserved symbol **10**
  - NARG reserved symbol **168**
  - nesting of structured directives **181**
  - NEXT directive (structured control) **179**
  - NLIST loader command **241 – 242**
  - NOERROR loader command **218**
  - NOFORMAT assembler directives **114**
  - NOLIST assembler directive **128**
  - noncommon section **187**
  - noncommon vs. common section attributes **75**
  - NOOBJ assembler directive **129**
  - NOPAGE assembler directive **130**
  - NOPAGE loader command **243**
  - numeric linker command arguments **201**
- O**
  - odd addresses **191**
  - odd memory locations, addressing **43**
  - OFFSET assembler directive **131 – 132**
  - OPEN librarian command **285**
  - operand field **4**
  - operand syntax and addressing modes **52**
  - operands **34**
  - operation field **4**
  - operators **16 – 17**
  - OPT assembler directive **133 – 138**
  - ORDER loader command **189, 244 – 246**
  - order of overlapping sections **245**
  - ORG assembler directive **139 – 140**
  - overlapping sections and section order **245**
- P**
  - PAGE assembler directive **141**
  - PAGE loader command **247**
  - page relocation **191**
  - parameter count (macros) **168**
  - parentheses, librarian command character **264**
  - PC memory indirect post-indexed mode (68020 model) **47**

- PC memory indirect pre-indexed mode (68020 model) **48**
- PC with 8-bit displacement and index mode (68000 model) **46**
- PC with 8-bit displacement and index mode (68020 model) **47**
- PC with base displacement and index mode (68020 model) **47**
- PC with displacement mode **46**
- PC, contents at execution time **43**
- PLEN assembler directive **142**
- plus sign, librarian command character **265**
- processing order, linker **201**
- program counter symbol
  - See* location counter (\*)
- program sections **75 – 80**
  - how the assembler assigns **80**
  - other things to know **79**
- PUBLIC loader command **248 – 249**
  
- Q**
  - qualifiers **24**
  - QUIT librarian command **280**
  
- R**
  - R flag in assembler listing **19**
  - REG assembler directive **143**
  - register direct modes **43**
  - registers **34 – 37**
    - floating-point **37**
  - relocatable expressions **83 – 84**
  - relocatable section **186**
  - relocatable vs. absolute symbols **82**
  - relocation **73 – 84**
  - relocation flags **19**
  - relocation types **191**
  - REPEAT directive (structured syntax) **177**
  - REPEAT...UNTIL loop **177**
  - REPLACE librarian command **286**
  - REPT assembler directive **144**
  - RESADD/RESMEM loader command **250 – 251**
  - reserved symbols **5, 10**
  - RESTORE assembler directive **145**
  - return codes **7**
    - librarian **266**
    - linker/loader error messages **193**

- ROM
  - initializing data from **226**
  - ROM, copying data from **226 – 228**
- S**
  - sample test program **266 – 267**
  - SAVE assembler directive **146**
  - SAVE librarian command **287 – 288**
  - SECT loader command **252**
  - SECT/SECTION assembler directive **147 – 148**
  - section attributes
    - common vs. noncommon **75**
    - how the assembler assigns **80**
    - short vs. long **76**
  - section types
    - attributes **77**
  - sections **75 – 80, 186 – 188**
    - names **187**
    - alignment **188**
    - attributes **187**
    - contents **188**
    - initialized data **226**
    - types **189**
    - types of **186**
  - SECTSIZE loader command **253**
  - semicolon, librarian command character **264**
  - semicolon, in linker command files **201, 203**
  - SET assembler directive **149**
  - severity, message **218**
  - shared data sections **187**
  - short section **187**
  - short vs. long section **76**
  - SIZEOF
    - generally **17**
  - SORDER loader command **189, 244 – 246**
  - SPC assembler directive **150**
  - SR register **35**
  - START loader command **254**
  - STARTOF
    - generally **17**
  - statement examples **5 – 6**
  - statements, assembler **3 – 4**
  - statically allocated data areas, A2-A5 relative addressing **65**

- status register **37**
  - MMU **35**
  - processor **35**
- structured control expressions **171 – 172**
- structured control statements **169 – 182**
- structured directive listings **182**
- structured directive nesting **181**
- subsections **187, 191**
- symbol names
  - HP format files **192**
- symbol types **11**
- symbols
  - assembler **9 – 11**
  - beginning with two question marks **9**
  - external **81**
  - forward defined, code generation for **61**
  - local **165 – 166**
  - location counter (\*) **11**
  - relocatable vs. absolute **82**
  - reserved **10**
  - valid examples **9**
- syntax for effective address fields **50 – 61**
- T** TTL assembler directive **151**
  - type (section) attributes **77**
- U** until directive (structured syntax) **177**
  - use of special characters **264**
  - user control of address modes **62 – 63**
- V** valid symbols, examples of **9**
  - variants of instruction types **33**
- W** WARN loader command **218**
  - warnings, librarian **266**
  - WHILE directive (structured syntax) **178**
  - WHILE...ENDW loop **178**
- X** XCOM assembler directive **152**
  - XDEF assembler directive **153**
  - XREF assembler directive **154 – 156**

## Index



---

## **Certification and Warranty**

---

### **Certification**

Hewlett-Packard Company certifies that this product met its published specifications at the time of shipment from the factory. Hewlett-Packard further certifies that its calibration measurements are traceable to the United States National Bureau of Standards, to the extent allowed by the Bureau's calibration facility, and to the calibration facilities of other International Standards Organization members.

---

### **Warranty**

This Hewlett-Packard system product is warranted against defects in materials and workmanship for a period of 90 days from date of installation. During the warranty period, HP will, at its option, either repair or replace products which prove to be defective.

Warranty service of this product will be performed at Buyer's facility at no charge within HP service travel areas. Outside HP service travel areas, warranty service will be performed at Buyer's facility only upon HP's prior agreement and Buyer shall pay HP's round trip travel expenses. In all other cases, products must be returned to a service facility designated by HP.

For products returned to HP for warranty service, Buyer shall prepay shipping charges to HP and HP shall pay shipping charges to return the product to Buyer. However, Buyer shall pay all shipping charges, duties, and taxes for products returned to HP from another country. HP warrants that its software and firmware designated by HP for use with an instrument will execute its programming instructions when properly installed on that instrument. HP does not warrant that the operation of the instrument, or software, or firmware will be uninterrupted or error free.

### **Limitation of Warranty**

The foregoing warranty shall not apply to defects resulting from improper or inadequate maintenance by Buyer, Buyer-supplied software or interfacing, unauthorized modification or misuse, operation outside of the environment specifications for the product, or improper site preparation or maintenance.

**No other warranty is expressed or implied. HP specifically disclaims the implied warranties of merchantability and fitness for a particular purpose.**

### **Exclusive Remedies**

**The remedies provided herein are buyer's sole and exclusive remedies. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory.**

Product maintenance agreements and other customer assistance agreements are available for Hewlett-Packard products.

For any assistance, contact your nearest Hewlett-Packard Sales and Service Office.