



***Microtec  
Research***

***ASM68K***  
***Documentation***  
***Set***

100113-009



***Microtec  
Research***

**2350 Mission College Blvd.**

**Santa Clara, CA 95054**

**Tel. 408.980.1300**

**Toll Free 800.950.5554**

**FAX 408.982.8266**



*Microtec*  
*Research*

***ASM68K***  
***User's***  
***Guide***

100010-008

## TRADEMARKS

Microtec® and Paragon® are registered trademarks of Microtec Research, Inc.  
MRI ASM™, MRI C™, MRI FORTRAN™, MRI Pascal™, MRI XRAY™, Source Explorer™,  
XHM302™, XRAY™, XRAY Debugger™, XRAY In-Circuit Debugger™, XRAY In-Circuit  
Debugger Monitor™, XRAY MasterWorks™, XRAY/MTD™, XRAY180™, XRAY29K™,  
XRAY51™, XRAY68K™, XRAY80™, XRAY86™, XRAY88K™, XRAY960™, XRAYG32™,  
XRAYH83™, XRAYH85™, XRAYM77™, XRAYSP™, XRAYTX™, and XRAYZ80™ are  
trademarks of Microtec Research, Inc.

Other product names mentioned in this document are trademarks or registered trademarks  
of their respective companies.

## RESTRICTED RIGHTS LEGEND

If this product is acquired under the terms of a: *DoD contract*: Use, duplication, or disclosure by  
the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of 252.227-7013.  
*Civilian agency contract*: Use, reproduction, or disclosure is subject to 52.227-19 (a) through (d)  
and restrictions set forth in the accompanying end user agreement. Unpublished rights reserved  
under the copyright laws of the United States.

MICROTEC RESEARCH, INCORPORATED  
2350 MISSION COLLEGE BLVD.  
SANTA CLARA, CA 95054

© Copyright 1988, 1989, 1990, 1991, 1992 Microtec Research, Inc. All rights reserved. No part of  
this publication may be reproduced, transmitted, or translated, in any form or by any means,  
electronic, mechanical, manual, optical or otherwise, without prior written permission of Microtec  
Research, Inc.

## Revision History

REV.	REVISION HISTORY	DATE	APPD.
-002	Updated from V 6.3 to V 6.4 the Reference Manual.	6/88	L.C.
-003	Updated from V 6.4 to V 6.5 the Reference Manual.	8/89	L.C.
-004	Printed at 80% for new packaging.	11/89	L.C.
-005	Printed V 6.4 at 80% for new packaging.	12/89	L.C.
-006	Updated from V 6.5 to V 6.6 the Reference Manual.	3/90	L.C.
-007	Updated to Version 6.8A.	9/91	L.L.
-008	Updated to Version 6.9	8/92	D.C.



## About This Manual

The *Microtec Research ASM68K Reference Manual* is written for the experienced program developer who is familiar with the Motorola 68000 family of microprocessors. It contains the following chapters and appendices which describe the relocatable macro assembler, linker, and object module librarian.

- Chapter 1 contains an introduction to the ASM68K software package. Key features of the ASM68K software as well as an overview are provided.
- Chapters 2 through 8 describe the ASM68K Assembler syntax and directives, and include several useful and informative program examples. Relocation, addressing modes, macros, and structured control statements are also explained in these chapters.
- Chapters 9 through 11 describe LNK68K Linker concepts, commands, and a sample linker session with input command files and output listings.
- Chapters 12 through 14 describe LIB68K Object Module Librarian concepts, commands, and example sessions.
- Appendix A, *ASCII and EBCDIC Codes*, contains hexadecimal values of ASCII and EBCDIC characters.
- Appendix B, *Assembler Error Messages*, describes the warning and error messages produced by the assembler.
- Appendix C, *Linker Error Messages*, describes the warning and error messages produced by the linker.
- Appendix D, *Librarian Error Messages*, describes the warning and error messages produced by the librarian.
- Appendix E, *Glossary*, contains a glossary of some assembler terms.
- Appendix F, *Object Module Formats*, describes output module formats generated by the linker.

- Appendix G, *C++ Support*, describes how the assembler, linker, and librarian were modified to support C++.

## Related Publications

The *Microtec Research ASM68K User's Guide* is written for the experienced program developer and assumes the developer has a working knowledge of the 68000 family of microprocessors. Although it provides several useful and informative program examples, this documentation does not describe the microprocessor itself. For such information, refer to:

- *Motorola Programmer's Reference Manual*, M68000PM/AD REV1
- *The Motorola M68000 Resident Structured Assembler Reference Manual*, M68KMASM/D8
- *Motorola 68000 Microprocessor User's Manual*, M68000UM(AD4)
- *Motorola 68020 Microprocessor User's Manual*, MC68020UM/AD REV 1
- *Motorola MC68881/MC68882 Floating-Point Coprocessor User's Manual*, MC68881UM/AD REV 1
- *Motorola 68030 Enhanced 32-Bit Microprocessor User's Manual*, MC68030UM/AD
- *MC68040 Enhanced 32-Bit Third Generation Microprocessor User's Manual*, MC68040UM/AD
- *Motorola 68851 Paged Memory Management Unit User's Manual*, MC68851UM/AD
- *Motorola CPU Central Processor Unit Reference Manual*, CPU32RM/AD
- *Motorola M68000 Family Reference*, M68000FR/AD
- *Motorola M68040 Microprocessor User's Manual*, M68040UM/AD

This manual assumes that you are familiar with programming and with the C programming language. For general information about C, refer to the following publications:

- *C: A Reference Manual*, 3rd ed., by Samuel P. Harbison and Guy L. Steele Jr., Prentice-Hall, Inc., 1991.

- *The C Programming Language*, 2nd ed., Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1988.

For information about other Microtec Research 68K toolkit components, refer to the following publications:

- *MCC68K Documentation Set*, Microtec Research, Inc.  
This documentation set describes how to use the MCC68K ANSI C Cross Compiler.
- *XRAY68K Documentation Set*, Microtec Research, Inc.  
This documentation set describes how to use the XRAY68K Debugger.
- *CCC68K Documentation Set*, Microtec Research, Inc.  
This documentation set describes how to use the CCC68K C++ Compiler.
- *C++ Capsule Class Library*, Microtec Research, Inc.  
This documentation set describes how to use the reusable data structure classes provided with the C++ Capsule Class Library.
- *XRAY68K Documentation Set*, Microtec Research, Inc.  
This documentation set describes how to use the XRAY68K Debugger.
- *XDM68K XRAY In-Circuit Debugger Monitor Documentation Set*, Microtec Research, Inc.  
This documentation set describes how to use the XDM68K In-Circuit Debugger Monitor with XRAY.
- *Microtec Research Flexible License Manager Documentation Set*, Microtec Research, Inc.  
This documentation set describes how to install and manage licenses for Microtec Research products that use Flexible Licensing.

## Notational Conventions

The *Microtec Research ASM68K Reference Manual* uses the conventions shown in Table P-1 (unless otherwise noted).

Table P-1. Notational Conventions

Symbol	Name	Usage
{ }	Curly Braces	Encloses a list from which you must choose an item.
[ ]	Square Brackets	Encloses items that are optional.
...	Ellipsis	Indicates that you may repeat the preceding item zero or more times.
	Vertical Bar	Separates items in a list.
	Punctuation	Punctuation other than that described here must be entered as shown.
	Typewriter font	Represents user input in interactive examples.
	<i>Italics</i>	Indicates a descriptive item that should be replaced with an actual item.

## Microprocessor References

The 68000 family of microprocessors includes the 68000, 68008, 68010, 68012, 68020, 68030, 68302, CPU32, 68040, 68881, 68882, 68851, 68HC000, 68HC001, 68EC000, 68EC020, 68EC030, 68EC040 and the CPU32 family: 68330, 68331, 68332, 68333, 68340. In this manual, they are referred to collectively as 68000.

## Questions and Suggestions

To help us respond to questions or suggestions regarding this product, we have provided two response forms in the back of this documentation.

The first form is a Reader's Response sheet, which is used to help us correct and improve our documentation. We always appreciate your comments, and by completing this form, you can participate directly in the revisions of our publications. The Reader's Response sheet is directed to Technical Publications.

The second form is a Software Performance Report, which should be completed if you encounter any errors or problems with Microtec Research software. This report is directed to Technical Support.



# Contents

---

## Preface

About This Manual .....	v
Related Publications .....	vi
Notational Conventions .....	viii
Microprocessor References .....	viii
Questions and Suggestions .....	viii

## 1 The Assembler

Overview .....	1-1
ASM68K Features .....	1-1

## 2 Assembly Language

Introduction .....	2-1
Assembler Statements .....	2-1
Statement Examples .....	2-2
Instruction Statement .....	2-2
Directive Statement .....	2-3
Macro Statement .....	2-3
Comment Statement .....	2-4
Symbolic Addressing .....	2-4
Assembly Time Relative Addressing .....	2-5
Assembler Syntax .....	2-6
Character Set .....	2-6
Symbols .....	2-7
Reserved Symbols .....	2-8
Relocatable Symbols .....	2-9
Assembly Program Counter .....	2-10
Constants .....	2-11
Integer Constants .....	2-11
Floating-Point Constants .....	2-12
Character Constants .....	2-13
Expressions .....	2-15
.STARTOF. and .SIZEOF. Operators .....	2-18
.STARTOF. Operator .....	2-18
.SIZEOF. Operator .....	2-19

## 3 Instructions and Address Modes

Introduction .....	3-1
Motorola Compatibility .....	3-1

Variants of Instruction Types .....	3-2
Instruction Operands .....	3-3
Registers .....	3-3
Addressing Modes .....	3-7
Register Direct Modes .....	3-9
Address Register Direct .....	3-9
Data Register Direct .....	3-9
Memory Addressing Modes .....	3-9
Address Register Indirect .....	3-9
Address Register Indirect with Postincrement .....	3-9
Address Register Indirect with Predecrement .....	3-9
Address Register Indirect with 16-bit Displacement .....	3-9
Address Register Indirect with 8-Bit Displacement and Index .....	3-10
Address Register Indirect with Base Displacement and Index .....	3-10
Memory Indirect Post-Indexed .....	3-10
Memory Indirect Pre-Indexed .....	3-11
Memory Reference Modes .....	3-11
Absolute Short .....	3-11
Absolute Long .....	3-12
Program Counter Indirect with 16-bit Displacement .....	3-12
Program Counter Indirect with 8-bit Displacement .....	3-12
Program Counter Indirect with Base Displacement and Index .....	3-12
Program Counter Memory Indirect Post-Indexed .....	3-12
Program Counter Memory Indirect Pre-Indexed .....	3-13
Non-Memory Reference Modes .....	3-13
Immediate .....	3-13
68881/882 Floating-Point and 68851 MMU	
Coprocessor and Addressing Modes .....	3-14
Assembler Syntax for Effective Address Fields .....	3-14
Addressing Mode Syntax .....	3-15
Operand Syntax and Addressing Modes .....	3-17
Dn Operand .....	3-17
An Operand .....	3-17
(An) Operand .....	3-17
(An)+ Operand .....	3-17
-(An) Operand .....	3-17
exp Operand .....	3-17
#exp Operand .....	3-17
(abs_exp,An) Operand .....	3-18
(Dn), (Rn,W), (Rn,L), (abs_exp,Dn), (abs_exp,Rn,W), and (abs_exp,Rn.L) Operands .....	3-18
(abs_exp,An,Rn{.W,L}) and (An,Rn{.W,L}) Operands .....	3-18
Square Brackets and No PC or ZPC .....	3-18
(rel_exp,Rn{.W,L}) Operand .....	3-19

---

.(rel_exp,An,Rn{.W.,L}) Operand .....	3-19
(exp,PC) Operand .....	3-19
(exp,PC,Rn{.W.,L}) Operand .....	3-20
Square Brackets and PC or ZPC .....	3-20
Addressing Mode Selection .....	3-20
ABS Section .....	3-21
abs_exp .....	3-21
External Reference in Specified Section .....	3-21
External Reference in Unspecified Section .....	3-21
rel_exp .....	3-21
unknown forward ref .....	3-21
REL Section .....	3-22
abs_exp .....	3-22
External Reference in Specified Section .....	3-22
External Reference in Unspecified Section .....	3-22
rel_exp .....	3-22
unknown forward ref .....	3-22
User Control of Addressing Modes .....	3-22
A2-A5 Relative Addressing .....	3-24
Address Register Indirect with Displacement Modes .....	3-24
Absolute Expressions versus Relocatable Expressions .....	3-24
The INDEX Linker Command .....	3-25
Accessing Statically Allocated Areas .....	3-25
Accessing Dynamically Allocated Areas .....	3-27
Example Listings .....	3-27
<b>4 Relocation</b>	
Introduction .....	4-1
Program Sections .....	4-1
Common versus Noncommon Attributes .....	4-2
Short versus Long Attributes .....	4-2
Section Alignment Attribute .....	4-3
Section Type Attributes .....	4-4
Section Types and HP 64000 Symbolic Files .....	4-5
Other Things to Know About Sections .....	4-6
How the Assembler Assigns Section Attributes .....	4-6
Linking .....	4-7
Relocatable versus Absolute Symbols .....	4-8
Relocatable Expressions .....	4-8
Complex Expressions .....	4-9
<b>5 Assembler Directives</b>	
Introduction .....	5-1
Assembler Directives .....	5-1

---

ALIGN — Specifies Instruction Alignment .....	5-4
CHIP — Specifies Target Microprocessor .....	5-5
COMLINE — Specifies Memory Space .....	5-8
COMMON — Specifies Common Section .....	5-9
DC — Defines Constant .....	5-11
DCB — Defines Constant Block .....	5-14
DS — Defines Storage .....	5-15
ELSEC — Specifies Conditional Assembly Converse .....	5-16
END — Ends Assembly .....	5-17
ENDC — Ends Conditional Assembly Code .....	5-19
ENDR — Ends Repeat .....	5-20
EQU — Equates a Symbol to an Expression .....	5-21
FAIL — Generates Programmed Error .....	5-22
FEQU — Equates a Symbol to a Floating-Point Expression .....	5-24
FOPT — Sets Floating-Point Options .....	5-26
FORMAT — Formats Listing .....	5-27
IDNT — Specifies Module Name .....	5-28
IFC — Checks if Strings Equal (Conditional Assembly) .....	5-29
IFDEF — Checks if Symbol Defined (Conditional Assembly) .....	5-31
IFEQ — Checks if Value Equal to Zero (Conditional Assembly) .....	5-32
IFGE — Checks if Value Non-Negative (Conditional Assembly) .....	5-34
IFGT — Checks if Value Greater Than Zero (Conditional Assembly) .....	5-35
IFLE — Checks if Value Non-Positive (Conditional Assembly) .....	5-36
IFLT — Checks if Value Less Than Zero (Conditional Assembly) .....	5-37
IFNC — Checks if Strings Not Equal (Conditional Assembly) .....	5-38
IFNDEF — Checks if Symbol Not Defined (Conditional Assembly) .....	5-40
IFNE — Checks if Value Unequal to Zero (Conditional Assembly) .....	5-41
INCLUDE — Includes Source File .....	5-42
IRP — Specifies Indefinite Repeat .....	5-43
IRPC — Specifies Indefinite Repeat Character .....	5-45
LIST — Generates Assembly Listing .....	5-47
LLEN — Sets Length of Line in Assembler Listing .....	5-48
MASK2 — Generates Code to Run on MASK2 (R9M) Chip .....	5-49
NAME — Specifies Module Name .....	5-50
NOOBJ — Does Not Create an Output Object Module .....	5-51
OFFSET — Defines Table of Offsets .....	5-52
OPT — Sets Options for Assembly .....	5-54
ORG — Begins Absolute Section .....	5-60
PAGE — Advances Listing Form to Next Page .....	5-61
PLEN — Sets Length of Listing Page .....	5-62
REG — Defines Register List .....	5-63
REPT — Specifies Repeat .....	5-64
RESTORE — Restores Options .....	5-65

SAVE — Saves Options .....	5-66
SECT / SECTION — Specifies Section .....	5-67
SET — Equates a Symbol to an Expression .....	5-69
SPC — Spaces Lines On Listing .....	5-70
TTL — Sets Program Heading .....	5-71
XCOM — Specifies Weak External Reference .....	5-72
XDEF — Specifies External Definition .....	5-73
XREF — Specifies External Reference .....	5-74
<b>6 Macros</b>	
Overview .....	6-1
Macro Heading .....	6-1
Macro Body .....	6-1
Macro Terminator .....	6-2
Calling a Macro .....	6-2
NARG Symbol .....	6-4
Macro Directives .....	6-5
ENDM — Terminates Macro Definition .....	6-6
LOCAL — Defines Local Symbol .....	6-7
MACRO — Enters Macro Definition .....	6-9
MEXIT — Exits Macro .....	6-11
<b>7 Structured Control Directives</b>	
Introduction .....	7-1
Structured Control Expressions .....	7-1
Loop Controls .....	7-4
Structured Control Directives .....	7-5
BREAK — Exits Loop Prematurely .....	7-6
FOR...ENDF — Loops Based on Counter .....	7-7
IF...THEN...ELSE...ENDI — Performs Conditional Execution .....	7-9
NEXT — Proceeds to Next Loop Iteration .....	7-11
REPEAT...UNTIL — Loops Until Condition Not True .....	7-12
WHILE...ENDW — Loops While Condition True .....	7-13
Structured Directive Nesting .....	7-14
Structured Directive Listings .....	7-14
<b>8 Sample Assembler Session</b>	
Introduction .....	8-1
Assembler Listing .....	8-1
Cross Reference Table Format .....	8-3
Sample Program Listing .....	8-4
The Object Module .....	8-7
<b>9 Linker Operation</b>	
Overview .....	9-1

---

Linker Features .....	9-2
Program Sections .....	9-2
Absolute Sections .....	9-3
Relocatable Sections .....	9-3
Common Section .....	9-4
Noncommon Section .....	9-4
Short Section .....	9-4
Long Section .....	9-5
Section Type .....	9-5
Section Alignment .....	9-5
Section Type .....	9-6
Memory Space Assignment .....	9-6
Relocation Types .....	9-8
Incremental Linking .....	9-9
Symbols in Linker Commands .....	9-10
Continuation and Escape .....	9-11

## 10 Linker Commands

Introduction .....	10-1
Command Syntax .....	10-1
Command Position Dependencies .....	10-3
Linker Commands .....	10-5
<b>ABSOLUTE</b> — Specifies Sections to Include in Absolute File .....	10-7
<b>ALIAS</b> — Specifies Section Assumed Name .....	10-9
<b>ALIGN</b> — Sets Alignment for Named Section .....	10-11
<b>ALIGNMOD</b> — Sets Alignment for Module Sections .....	10-13
<b>BASE</b> — Specifies Location to Begin Loading .....	10-14
<b>CASE</b> — Controls Case-Sensitivity .....	10-16
<b>CHIP</b> — Specifies Target Microprocessor .....	10-18
<b>Comment</b> — Specifies Linker Comment .....	10-21
<b>COMMON</b> — Sets Common Section Load Address .....	10-22
<b>CPAGE</b> — Sets Common Section to be Page Relocatable .....	10-24
<b>DEBUG_SYMBOLS</b> — Retains or Discards Internal Symbols .....	10-26
<b>END</b> — Ends Command Stream and Finishes Linking .....	10-27
<b>ERROR</b> — Modifies Message Severity .....	10-28
<b>EXIT</b> — Exits Linker Without Linking .....	10-29
<b>EXTERN</b> — Creates External References .....	10-30
<b>FORMAT</b> — Selects Output Format .....	10-31
<b>INCLUDE</b> — Includes a Command File .....	10-32
<b>INDEX</b> — Specifies Run-Time Value of Register An .....	10-33
<b>INITDATA</b> — Specifies Initialized Data in ROM .....	10-35
<b>LISTABS</b> — Lists Symbols to Output Object Module .....	10-37
<b>LISTMAP</b> — Specifies Layout and Content of the Map .....	10-38
<b>LOAD</b> — Loads Object Modules and/or Library Modules .....	10-40

LOAD_SYMBOLS — Loads Symbol Information of Specified Object Modules .....	10-42
LOWERCASE — Shifts Names to Lowercase .....	10-43
MERGE — Combines Named Module Sections .....	10-45
NAME — Specifies Output Module Name .....	10-47
ORDER — Specifies Long Section Order .....	10-48
PAGE — Sets Page Alignment .....	10-50
PUBLIC — Specifies Public Symbols (External Definitions) .....	10-52
RESADD — Reserves Regions of Memory .....	10-54
RESMEM — Reserves Regions of Memory .....	10-55
SECT — Sets Section Load Address .....	10-56
SECTSIZE — Sets Minimum Section Size .....	10-57
SORDER — Specifies Short Section Order .....	10-58
START — Specifies Output Module Starting Address .....	10-60
SYMTRAN — Transforms Public/External Symbols .....	10-61
UPPERCASE — Shifts Names to Uppercase .....	10-63
WARN — Modifies Message Severity .....	10-64

## 11 Sample Linker Session

Introduction .....	11-1
Linker Listings .....	11-1
Sample Linker Listing .....	11-4
Linker Command File .....	11-4
Linker Map File .....	11-5
Linker Absolute File .....	11-9
Assembly Listings for the Linker Example .....	11-10
Assembly Listing 1 .....	11-11
Assembly Listing 2 .....	11-14
Assembly Listing 3 .....	11-17

## 12 Librarian Operation

Introduction .....	12-1
Librarian Function .....	12-1
Return Codes .....	12-6

## 13 Librarian Commands

Introduction .....	13-1
Command Syntax .....	13-1
Blanks .....	13-2
Command File Comments .....	13-2
Command Summary .....	13-3
ADDLIB — Adds Module(s) from Another Library .....	13-4
ADDMOD — Adds Object Module(s) to Current Library .....	13-5
CLEAR — Clears Library Session Since Last SAVE .....	13-6

---

CREATE — Creates New Library .....	13-7
DELETE — Deletes Module(s) from Current Library .....	13-8
DIRECTORY — Lists Library Modules .....	13-9
END / QUIT — Terminates Librarian Execution .....	13-10
EXTRACT — Copies Library Module to a File .....	13-11
FULLDIR — Displays Library or Library Module Contents .....	13-12
HELP — Displays Context-Sensitive Command Syntax .....	13-14
OPEN — Opens an Existing Library .....	13-15
REPLACE — Replaces Library Module .....	13-16
SAVE — Saves Contents of Current Library .....	13-17

## 14 Sample Librarian Session

Overview .....	14-1
Librarian Sample Program 1 .....	14-1
Librarian Sample Program 2 .....	14-4

## Appendix A: ASCII and EBCDIC Codes .....

A-1

## Appendix B: Assembler Error Messages

Introduction .....	B-1
Assembler Messages and Errors .....	B-2

## Appendix C: Linker Error Messages

Introduction .....	C-1
Message Severity Levels .....	C-1
Linker Messages and Errors .....	C-2

## Appendix D: Librarian Error Messages

Introduction .....	D-1
Message Severity Levels .....	D-1
Librarian Error Messages .....	D-2

## Appendix E: Glossary .....

E-1

## Appendix F: Object Module Formats

S-Record Format .....	F-1
Module Record (Optional) .....	F-1
Symbol Record (Optional) .....	F-1
Header Record .....	F-2
Data Record .....	F-2
Record Count Record .....	F-4
Sample S-Record .....	F-6

## Appendix G: C++ Support

Overview .....	G-1
----------------	-----

Name Mangling and Demangling .....	G-1
ASM68K Assembler .....	G-1
LNK68K Linker .....	G-2
Demangling Example .....	G-2
Symbol Name Length .....	G-3
LIB68K Librarian .....	G-4

---

## Figures

---

Figure 2-1. Floating-Point Representation With and Without Underscores .....	2-13
Figure 3-1. Absolute versus Indirect Addressing Modes .....	3-26
Figure 3-2. A2-A5 Relative Addressing Example .....	3-28
Figure 3-3. Using the INDEX Command with Offset .....	3-30
Figure 3-4. Using the INDEX Command without Offset .....	3-32
Figure 8-1. Sample Assembly Listing .....	8-4
Figure 11-1. Linker Command File .....	11-4
Figure 11-2. Linker Map File .....	11-5
Figure 11-3. Linker Absolute File .....	11-9
Figure 11-4. Assembly Listing 1 .....	11-11
Figure 11-5. Assembly Listing 2 .....	11-14
Figure 11-6. Assembly Listing 3 .....	11-17
Figure 12-1. Three Relocatable Object Modules Resulting from Assembly .....	12-2
Figure 12-2. Relocatable Object Module in MAIN.OBJ .....	12-4
Figure 12-3. MAIN Module Modified to Call SINCOS Module .....	12-4
Figure 14-1. Librarian Sample Program 1 Output Listing .....	14-2
Figure 14-2. Librarian Sample Program 2 Output Listing .....	14-5
Figure F-1. Data Record Format for 16-Bit Load Address .....	F-3
Figure F-2. Record Count Record Format .....	F-4
Figure F-3. Terminator Record Format for 24-Bit Load Address .....	F-5
Figure G-1. C++ Program Listing .....	G-1

## Tables

---

Table P-1.	Notational Conventions .....	viii
Table 2-1.	Special Characters Recognized by the Assembler .....	2-7
Table 2-2.	Reserved Symbols .....	2-8
Table 2-3.	Constant Value Ranges .....	2-11
Table 2-4.	Descriptors Used for Constants .....	2-12
Table 2-5.	Character Constant Field Sizes .....	2-14
Table 2-6.	Summary of Operators and their Precedence .....	2-17
Table 3-1.	Summary of Variants of Instruction Types .....	3-2
Table 3-2.	Summary of Register Mnemonics .....	3-3
Table 3-3.	Addressing Modes .....	3-7
Table 3-4.	Assembler Syntax for Effective Address Fields .....	3-14
Table 4-1.	ASM68K/HP 64000 Section Mapping .....	4-5
Table 4-2.	How the Assembler Assigns Section Attributes .....	4-7
Table 5-1.	Alphabetical listing of the Assembler Directives .....	5-1
Table 5-2.	Processor Identification .....	5-5
Table 6-1.	Alphabetical Listing of the Macro Directives .....	6-5
Table 7-1.	Conditional Code Comparisons .....	7-3
Table 7-2.	Alphabetical Listing of the Structured Control Directives .....	7-5
Table 9-1.	16-Bit Absolute Address Memory Areas .....	9-5
Table 9-2.	Section Types .....	9-6
Table 10-1.	Representation of Numeric Command Arguments .....	10-1
Table 10-2.	Linker Commands .....	10-5
Table 10-3.	68000 Family Memory Addressing .....	10-19
Table 13-1.	Special Characters Recognized by the Librarian .....	13-1
Table 13-2.	Librarian Commands and Abbreviations .....	13-3
Table A-1.	ASCII and EBCDIC Codes .....	A-1
Table C-1.	Linker Message Severity Levels .....	C-2
Table D-1.	Librarian Message Severity Levels .....	D-2

# The Assembler 1

---

## Overview

The Microtec Research ASM68K Assembler for the 68000 microprocessor translates symbolic machine instructions into binary object code that can be executed by a 68000 microprocessor.

The ASM68K Assembler is a two-pass program that issues helpful error messages, produces an easy-to-read program listing and symbol table, and generates a binary relocatable object (link) module. The linker produces an output object module file in IEEE-695 or Motorola S-record format. Both global and local symbols can be included in the output object module file for symbolic debugging with the Microtec Research XRAY68K Debugger.

The ASM68K mnemonic operation codes, the assembler directives, and the assembler syntax are compatible with that used by Motorola in its software products and documentation. The assembler produces object code in a relocatable format. Multiple relocatable modules are then typically linked into a single absolute module by the LNK68K Linker.

Relocatable object modules can be grouped in libraries using the object module librarian, LIB68K. The library modules can then be linked with other relocatable modules using LNK68K. Only those library modules required to resolve external references will be linked into the final object module.

## ASM68K Features

Significant features of ASM68K include:

- 68000/08/10/12/20/30/40, 68851/881/882, 68HC000/01, 68EC000/20/30/40 and CPU32 family instructions supported
- 68881/82 floating-point instructions
- 68851 paged MMU instructions
- Versatile directive set
- Address constants can be easily specified
- Assembly-time relative addressing
- Several different data creation statements
- Storage reservation statements
- Character codes can be specified in ASCII or EBCDIC

- Flexible assembly listing control statements
- Comments and remarks can be used for documentation
- Manufacturer-compatible symbolic machine operation codes (opcodes, directives)
- Symbolic and relative address assignments and references
- Symbol or cross-reference table listings
- Absolute object modules can be produced in Motorola S-record, IEEE-695, or HP-OMF format
- Absolute or relocatable (incremental linking) object modules generated by the linker
- Complex expression evaluation
- Unlimited forward symbolic references
- Conditional assembly facility
- User-defined macro facility
- Run-time structured loop control directives
- Symbols in the output object module for symbolic debugging
- Complex relocation in the linker
- High-level debugging information for the Microtec Research XRAY68K Debugger
- Symbols can be made case-sensitive or case-insensitive
- C++ support

These features will help you to produce well-documented, modular, working programs.

# Assembly Language 2

---

## Introduction

A 68000 microprocessor executable program consists of a sequence of binary numbers contained in memory. These numbers represent 68000 instructions, memory addresses, and data. It is possible to program the microprocessor by manually calculating and encoding the numbers that cause the microprocessor to perform the desired functions. However, this method is very slow and tedious. An assembler provides an easier method of writing programs by allowing machine instructions to be encoded symbolically with English-like mnemonics and symbols.

This chapter describes the format of assembly language source files and assembler statements.

## Assembler Statements

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

1. Instruction statements
2. Directive statements
3. Macro statements
4. Comment statements

The syntax for instructions, directives, macros, and comments is shown below:

### Syntax:

*label    operation    operand(s)    ;comment*

### Description:

The components of an assembler statement are defined below.

*label*

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. No spaces can be placed between the label and the colon.
- Column one when the colon is omitted.

A label can be the only field in a statement. The first 127 characters of a label are significant.

Labels are case-sensitive by default. Case-insensitivity can be turned on by using the **OPT NOCASE** assembler directive.

*operation*      The operation field specifies a symbolic operation code, a directive, or a macro call. If present, this field must either begin after column one or be separated from the label field by one or more blanks, tabs, or a colon.

*operand*      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*      The comment field provides a place to put a message 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 a semicolon, asterisk, or exclamation point.

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 can be a maximum of 512 characters long.

## Statement Examples

The following sections show examples of the types of assembly statements.

### Instruction Statement

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

**Example:**

ISAM MOVE MEM,D2

**where:**

ISAM	Is a symbol ( <i>label</i> ) representing the memory address of the instruction.
MOVE	Is a symbolic opcode ( <i>operation</i> ) representing the bit pattern of the load instruction.
MEM	Is a symbol ( <i>operand</i> ) representing a memory address.
D2	Is a reserved symbol (part of <i>operand</i> ) representing data register number 2.

**Directive Statement**

A directive statement is a control statement to the assembler. It is not translated into a machine instruction, and its operation field always begins with a period (.). For example:

**Example:**

ABAT DC DELT

**where:**

ABAT	Is a symbol ( <i>label</i> ). The assembler will assign the value of the current assembly program counter to this symbol. The assembly program counter contains the address of the first byte of the code generated by the directive .DATA.
DC	Is a directive ( <i>operation</i> ) that instructs the assembler program to allocate two bytes of memory.
DELT	Is a symbol ( <i>operand</i> ) representing an address. The address will be placed into the two bytes allocated by the .DATA directive.

**Macro Statement**

A macro statement is a call to a sequence of instructions. The call can be made many times from any part of the program. The requirements for creating macros are described in the Chapter 6, *Macros*. The macro call has the same format as a directive statement, except that the operator is one of the macro commands.

**Example:**

```
M1    MACRO X, Y
```

**where:**

- |       |  |
|-------|--|
| M1    | Is the name used to call the macro from other parts of the program.  |
| MACRO | Is a directive that assigns a name and a formal set of parameters to the operands in the statement. This directive also indicates that all of the statements following statements, up to the occurrence of the ENDM directive, are part of the macro definition. |
| X, Y  | Are symbols that represent parameters passed to the macro.   |

**Comment Statement**

A comment statement is not processed by the assembler. Instead, it is reproduced on the assembly listing and can be used to document groups of assembly language statements. Comment statements are preceded by a semicolon, asterisk, or exclamation point.

**Example:**

- \* This is a comment statement.
- : This is another comment statement.
- ; This is another comment statement.

Blank lines are also treated as comment statements.

**Note**

An asterisk is considered a comment statement only if it is in the first column. In any other column, the asterisk may be interpreted as the assembly program counter or multiplication operator.

## Symbolic Addressing

When writing statements in assembler language, the machine operation code is usually expressed symbolically. For example, the machine instruction that rotates the contents of Data Register 1 to the left one bit can be expressed as:

```
ROL #1,D1
```

When translating this symbolic operation code and its arguments into machine code for the 68000, the assembler defines two bytes containing \$E3 and \$59 at the

memory location indicated by the current assembly program counter. The assembly program counter is an internal variable kept by the assembler that is always set to the address of the byte currently being assembled.

You can optionally attach a label to such an instruction.

**Example:**

```
SAVR ROL #1,D1      ; SAVR is a label
```

Whenever there is a valid symbol in the label field, the assembler assigns the address contained in the assembly program counter to the symbol. In the given example, if the ROL instruction is to be stored at the address 127, then the symbol SAVR will be set to the value 127 for the duration of the assembly.

The symbol could then be used anywhere in the source program to refer to the instruction location. The important concept is that the address of the instruction need not be known; only the symbol need be used to refer to the instruction location.

Therefore, when jumping to the ROL instruction, you could write:

```
JMP SAVR
```

When the jump instruction is translated into machine code by the assembler, the address of the ROL instruction is placed in the address field of the instruction.

## Assembly Time Relative Addressing

Symbolic addresses can be used to refer to nearby locations without defining new labels. This form of addressing is called relative addressing and may be accomplished through the use of the plus (+) and minus (-) operators.

**Example:**

PTAB	EQU	\$F37
BEG	JMP	BEG+6
	MOVE	D1,D2
	CMPI.B	#\$2F,(A3)
	ROL	#2,D1
END	ADDQ	#7,PTAB

In this example, the instruction JMP BEG+6 refers to the ROL instruction. BEG+6 means the address of BEG plus 6 bytes.

This type of addressing is not recommended as the variation in the number of bytes per instruction will tend to cause references to the wrong location. In the above example, for instance, the MOVE instruction requires 2 bytes and the CMPI instruction requires 4 bytes.

The expansion of the above example is shown as follows:

00000F37	PTAB	EQU	\$F37
00000000 4EFA 000A 4E71		JMP	BEG+6
00000006 3401	BEG	MOVE	D1,D2
00000008 0C13 002F		CMPI.B	#\$2F,(A3)
0000000C E559		ROL	#2,D1
0000000E 5E78 0F37		ADDQ	#7,PTAB
	END		

## Assembler Syntax

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

### Character Set

The assembler recognizes the alphabetical characters A-Z and a-z, the numeric characters 0-9, and the special characters shown in Table 2-1. Any other characters, except those in a comment field or a string, generate an error. Many of the special characters have no previously defined meanings except as character constants.

The characters are not case-sensitive by default. If case-sensitivity is turned on, the case is maintained.

**Table 2-1. Special Characters Recognized by the Assembler**

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

## Symbols

Symbols are used to represent arithmetic values, memory addresses, bit arrays (masks), etc. A symbol is a sequence of characters. The first 127 characters of a symbol are significant. The first character in a symbol must be alphabetic or one of three special characters:

1. ? question mark
2. . period
3. \_ underscore

Subsequent characters in the symbol can consist of the special characters just mentioned, alphabetic letters, or numeric digits. Embedded blanks are not permitted in symbols. Symbols can be made case-insensitive by using the **OPT NOCASE** directive.

Avoid using symbols that start with two underscores, since the assembler uses this notation for its own "local" symbols.

The assembler treats symbols beginning with two or more question marks (e.g., ??LAB1) as assembler-generated symbols. When the assembler creates unique labels in macro expansions, it generates symbols in the form ??0001, ??0002, and so on. These assembler-generated symbols are not included in the assembler listing

or in the HP **asmb\_sym** file unless the **OPT G** assembler flag is set. If you do code your own symbols beginning with two question marks, these symbols are not available for debugging unless you specify the **OPT G** directive.

The assembler generates local symbols in macros that start with the character sequence **\@**. These symbols are only valid inside a macro.

Symbols are used to represent arithmetic values, memory addresses, bit arrays (masks), etc.

### **Examples of valid symbols:**

```
LAB1
mask
LOOP$NUM
LOOP_COUNT
L23456789012345678901234567890123456789
```

### **Examples of invalid symbols:**

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

## **Reserved Symbols**

The ASM68K Assembler contains several reserved symbols, or keywords, that you cannot redefine. These reserved symbols, shown in table Table 2-2, are the symbolic register names used to denote the various hardware registers; they will not appear in a symbol table or in a cross-reference listing.

**Table 2-2. Reserved Symbols**

<b>Category</b>	<b>Symbol</b>
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
68010 Registers	VBR, SFC or SFCR, DFC or DFCR
68020 Registers	ZA0, ZA1, ZA2, ZA3, ZA4, ZA5, ZA6, ZA7, ZD0, ZD1, ZD2, ZD3, ZD4, ZD5, ZD6, ZD7, ZPC

**(cont.)**

Table 2-2. Reserved Symbols (cont.)

Category	Symbol
68881/82 Registers	FP0, FP1, FP2, FP3, FP4, FP5, FP6, FP7, CONTROL, STATUS, IADDR, FPCR, FPSR, FPIAR
68851 Registers *	VAL, CAL, SCC, CRP, SRP, DRP, TC, AC, PCSR, BAD0, BAD1, BAD2, BAD3, BAD4, BAD5, BAD6, BAD7, BAC0, BAC1, BAC2, BAC3, BAC4, BAC5, BAC6, BAC7
68EC030 Registers	CRP, SRP, TC, AC0, AC1, ACUSR
68030 Registers *	CRP, SRP, TC, TT0, TT1, MMUSR
68040 Registers *	DTT0, DTT1, ITT0, ITT1, URP

\* Reserved only during this mode.

You can also define keywords to represent these reserved symbols through the EQU directive.

#### Example:

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

The above example is the same as ADD.B #1, D4.

The reserved symbol NARG represents the number of arguments passed on a macro call.

#### Relocatable Symbols

Each ASM68K symbol has an associated symbol type that denotes the symbol as absolute or relocatable. If relocatable, the type also indicates the section where the symbol. Symbols whose values are not dependent upon program origin are called absolute symbols. Symbols whose values change when the program origin is changed are called relocatable symbols.

All external references are considered relocatable even though the external can be defined as absolute. When assembling a module, the assembler does not know whether an externally defined symbol in another module is relocatable or absolute and, therefore, assumes the external is relocatable.

Absolute and relocatable symbols can both appear in absolute or relocatable program sections. The characteristics of absolute and relocatable symbols are as follows:

A symbol is absolute when:

- It is in the label field of an instruction when the program is assembling an absolute section.
- It is made equal to an absolute expression by the EQU or SET directive, regardless of whether the program is assembling a relocatable section.
- It is an external reference with no section attached for the purpose of determining addressing modes.

A symbol is relocatable when:

- It is in the label field of an instruction when the program is assembling a relocatable section.
- It is made equal to a relocatable expression by the EQU or SET directive.
- It is an external reference with a section attached.
- It is a user-defined relocatable section name.
- A reference is made to the program counter (\$) while assembling a relocatable section.

## Assembly Program Counter

During the assembly process, the assembler maintains a variable that always contains the address of the current memory location being assembled. This variable is called the assembly program counter. It is used by the assembler to assign addresses to assembled bytes, but it is also available to the programmer. The asterisk (\*) is the symbolic name of the assembly program counter. It can be used like any other symbol, but it cannot appear in the label field.

### Example:

```
10    BRA $
```

The relative branch instruction is in location 10. The instruction directs the microprocessor to branch to the beginning of the current instruction (i.e., location 10). The program counter in this example contains the value 10 and the instruction will be translated to a relative jump to location 10 from location 10. This example is

useful when waiting for an interrupt. This example is useful when waiting for an interrupt.

The Assembly Program Counter should not be confused with the hardware Program Counter (PC). The PC always contains a value 2 bytes greater than the Assembly Program Counter. The PC contains the address of the next instruction to be executed.

## Constants

A constant is an invariant quantity. It can be an arithmetic value or a character code. Arithmetic values can 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. When a constant is negative, its equivalent two's complement representation is generated and placed in the field specified. The ranges per byte of constants are shown in Table 2-3.

**Table 2-3. Constant Value Ranges**

Number of Bytes	Value Range
1 (unsigned)	0 to 255
2 (unsigned)	0 to 65,535
4 (unsigned)	0 to 4,294,967,295
1 (two's complement)	-128 to +127
2 (two's complement)	-32,768 to +32,767
4 (two's complement)	-2,147,483,648 to +2,147,483,647

A number whose most significant bit is set can be interpreted either 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 use. The assembler will correctly recognize numbers in either form; however, you are 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, the assembler generates an error message.

Decimal constants are 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.

Table 2-4 shows the available descriptors and their meanings. If no descriptor is given, the default number base is used.

**Table 2-4. Descriptors Used for Constants**

Prefix	Constant	Suffix
%	binary	B
	decimal	D
@	octal	O, Q
\$	hexadecimal	H

### Examples:

The following examples show how integer constants can be specified.

```
t21_a    equ   $10101 ;binary
t21_b    equ   10101b ;binary
t21_c    equ   @25      ;octal
t21_d    equ   25o      ;octal
t21_e    equ   25q      ;octal
t21_f    equ   21       ;decimal
t21_h    equ   21d      ;decimal
t21_i    equ   $15      ;hex
t21_j    equ   15h      ;hex

dc.b    $10101 ;binary
dc.b    10101b ;binary
dc.b    @25      ;octal
dc.b    25o      ;octal
dc.b    25q      ;octal
dc.b    21       ;decimal
dc.b    21d      ;decimal
dc.b    $15      ;hex
dc.b    15h      ;hex
```

### Floating-Point Constants

Floating-point numbers can 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.

**Examples:**

```
3.14159
-22E-100 ; -22 * 10-100
```

Underscores can be placed before or after the E to increase readability. Underscores are ignored in determining the value of a constant as shown in Figure 2-1.

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 digits specified are placed in the field as they stand; you are 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 permitted only in the **DC**, **DCB**, and **FEQU** directives. Additional information about floating-point constants appears in the description of the **DC** directive in the *Assembler Directives* chapter of this manual.

Microtec Research ASM68K Version x.y				Wed Jul 22 13:27:87 1992	Page 1
Line Address					
1		SYM1	FEQU	:1234_5678_9ABC_DEF_	
2		SYM2	FEQU	:123456789ABCDEF	
3					
4	00000000 1234 5678 9ABC DEFB 0000 0000		DC.X	SYM1	
5	0000000C 1234 5678 9ABC DEFB 0000 0000		DC.X	SYM2	
6	00000018 1234 5678 9ABC DEFB 0000 0000	SYM3	DC.X	:1234_5678_9ABC_DEF_	;hexadecimal representation
7					
8	00000024 1234 5678 9ABC DEFB 0000 0000	SYM4	DC.X	:123456789ABCDEF	; of floating-point numbers ; using underscores
9	00000030 40EF 0000 C8E3 E8BA AD78 4E36	SYM5	DC.X	35E42	;decimal representation
10	0000003C 40EF 0000 C8E3 E8BA AD78 4E36	SYM6	DC.X	35_E_42	; of floating-point numbers ; using underscores
11					
12				END	

Figure 2-1. Floating-Point Representation With and Without Underscores

**Character Constants**

An ASCII or EBCDIC character constant can 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 you do not supply a descriptor, the default string format is ASCII.

EBCDIC character constants can only be used as operands of instructions, within expressions, or as operands of the storage allocation directives. Otherwise, the EBCDIC specification will be ignored.

A character constant consists of up to four characters and corresponds to the field sizes shown in Table 2-5.

**Table 2-5. Character Constant Field Sizes**

Field Size	Number of Characters
8-bit	1 maximum
16-bit	up to 2 maximum
32-bit	up to 4 maximum

Constants shorter than the length of the field are left-justified within the field, and the remainder of the field is filled with zeros. The size of the field is determined by the instruction word size which is "word" by default.

### Examples:

```

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

```

When character strings are used as operands of word and longword operations, the assembler assigns values according to the following rules which are compatible with the Motorola M68000 Family Resident Structured Assembler:

1. 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.

### Example:

```

DC.B 'A'           ; Hex value 41
DC.B 'AB'          ; Hex value 4100
DC.W 'A'           ; Hex value 4100
DC.W 'AB'          ; Hex value 4142
DC.W 'ABC'         ; Hex value 4142 4300
DC.L 'A'           ; Hex value 41000000
DC.L 'AB'          ; Hex value 41420000
DC.L 'ABC'         ; Hex value 41424300
DC.L 'ABCD'        ; Hex value 41424344
DC.L 'ABCDE'       ; Hex value 4142434445000000

```

2. 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.

**Example:**

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

You must use two single quotation marks to generate code for a single quotation mark in a character constant or string.

**Example:**

```
'DON' 'T'
```

The code for a single quotation 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. The maximum number of symbols in an expression is 48. Expressions are evaluated left to right, subject to the precedence of operators shown in Table 2-6. Parentheses can 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  $==$  operator is used to determine whether an operand exists. This is further described in the section *Macro Call* of Chapter 6, *Macros*.

The comparison operators  $=$ ,  $\geq$ , 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. Comparisons against 0 are not very useful.

**Example:**

```
IFEQ 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.

**Example:**

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

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. Whenever 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  
VAL1=VAL2  
IDAM&255
```

The comparison operators return 1 if the comparison is true and zero if the comparison is not true. Embedded blanks are allowed in expressions. The assembler interprets spaces as termination characters.

Table 2-6. Summary of Operators and their Precedence

Precedence	Operator	Meaning
1	=	test for existing operand
2	+	unary plus
	-	unary minus
	"	bit-wise logical not
	.SIZEOF.	size of section
	.STARTOF.	start of section
3	>>	shift right
	<<	shift left
4	&	bit-wise logical AND
	!	bit-wise logical OR
	!!	exclusive OR
5	*	multiplication
	/	division
6	+	addition
	-	subtraction
7	=, <>	equality, inequality
	>, >=	greater than, greater than or equal
7	<, <=	less than, less than or equal

## .STARTOF. and .SIZEOF. Operators

The **.STARTOF.** and **.SIZEOF.** operators help you to write code that initializes or copies logical sections of memory. When using the **.STARTOF.** and **.SIZEOF.** operators, the section that is being referenced should be previously defined with the **SECT** or **COMMON** directive within the same assembly file. All **.STARTOF.** and **.SIZEOF.** values are resolved at link time. If this is not done, the assembler will create the section and its combine type; this combine type may not be desirable and could cause section mismatch errors at link time.

For example, if your assembly file contains an instruction that creates a non-common section named "stack":

```
move #.SIZEOF.(stack),D0
```

and the linker command file contains a directive that creates a common section named "stack":

```
common stack = $1FF
```

the linker will generate a "section mismatch" error at link time. To solve this problem, create the common section "stack" before the **.SIZEOF.** operator is used:

```
COMMON      stack  
...  
SECT       code  
...  
move       #.SIZEOF.(stack),D0
```

## .STARTOF. Operator

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

### Syntax:

```
.STARTOF.(section_name)
```

### Description:

*section\_name*      The name of a section.

## .SIZEOF. Operator

The .SIZEOF. operator gives the size in the number of bytes of the combined section.

### Syntax:

.SIZEOF. (*section\_name*)

### Description:

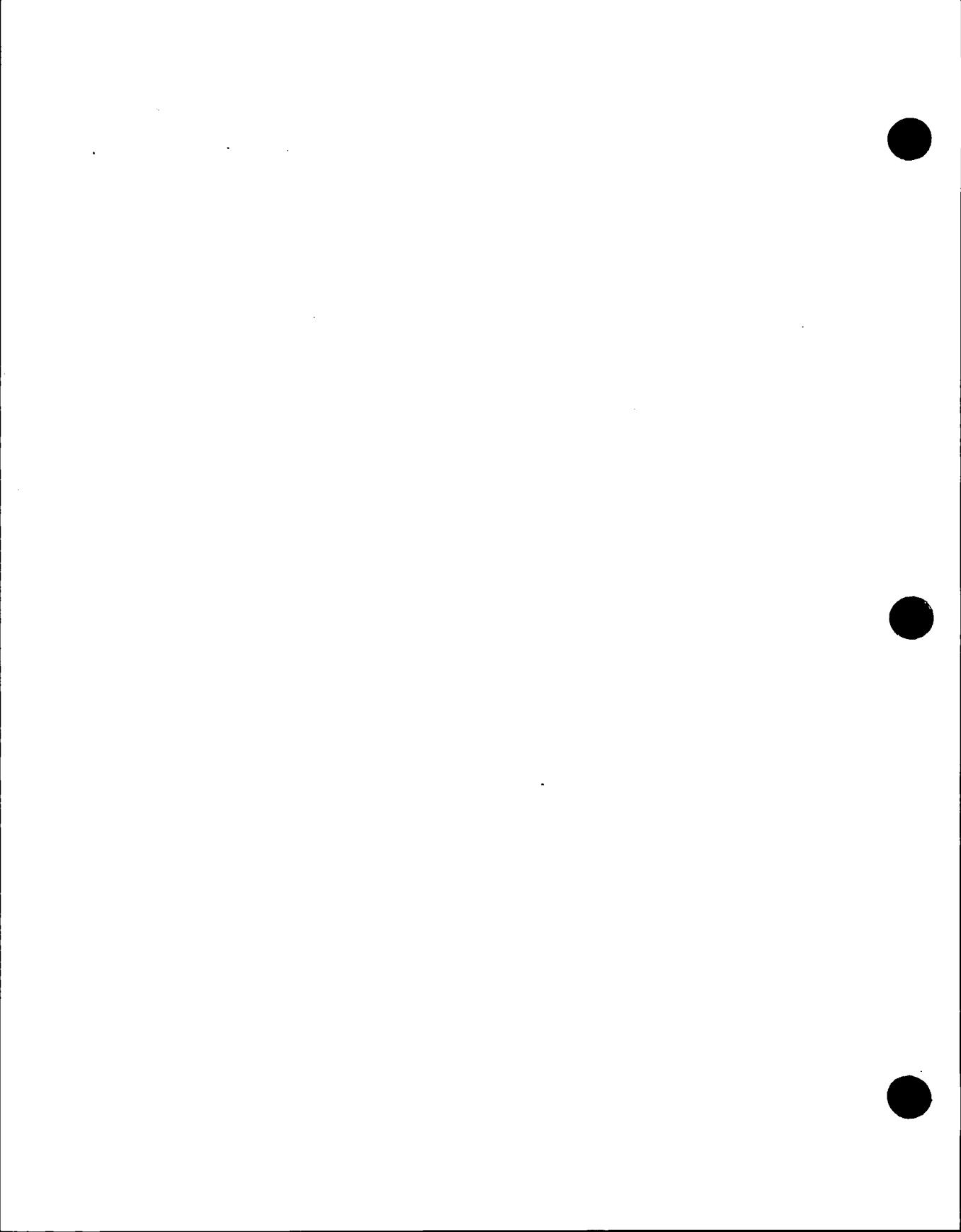
*section\_name*      The name of a section.

Each .STARTOF. and .SIZEOF. operator gives a relocatable value.

### Example:

\* This routine will clear memory for the full address  
\* range of the final combined section that contains  
\* the subsection zerovals from this module.

```
OPT CASE,D
SECT zerovals,,D
DS.B $300
SECT code,,C
LEA .STARTOF.(zerovals),A0 ; start of section
MOVE #.SIZEOF.(zerovals),D1; length of section
initsect: CLR.B (A0)+ ; clear the address
SUBQ #1,D1 ; decrement counter
BGT initsect ; continue looping until count=0
RTS ; otherwise return to calling routine
END
```



# Instructions and Address Modes 3

---

## Introduction

This chapter describes:

- The 68000/08/10/12/20/30/40, 68EC000/20/30/40, 68851/881/882, 68HC000/01, and CPU32 assembly language instruction mnemonics and qualifiers.
- How the assembler will generate code for variants of certain instructions depending on the instruction's operands.
- The addressing modes for the 68000/08/10/12/20/30/40, 68HC000/01, 68EC000/20/30/40, 68851/881/882 and CPU32 family processors.
- Assembler syntax and the addressing modes that are generated for a particular syntax.
- How you can control generation of addressing modes by setting or clearing various assembler options (with the OPT directive).

The 68020 and 68030 addressing modes and memory ranges are the same as those of the 68040. The 68020 mnemonics with the exception of CALLM and RTM, however, are a subset of the 68030.

## Motorola Compatibility

In some cases, the Motorola assembler manual and the Motorola processor manuals define different mnemonics for the same operation. Wherever possible, ASM68K recognizes both sets of mnemonics. However, the following case cannot be reconciled:

```
DIVS.L <ea>,Dq ;Dq is both upper and lower half of  
;64-bit dividend  
DIVU.L <ea>,Dq ;Dq is both upper and 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 upper-half and the lower-half of the number which is not a useful operation.

The ASM68K Assembler's behavior contradicts the description in Motorola's

*MC68020 32-bit Microprocessor User's Manual.* However, ASM68K behaves consistently with the documented *M68000 Family Resident Structured Assembler*.

To divide a 32-bit dividend and obtain a 32-bit quotient, write the following instruction:

```
DIVSL.L <ea>,Dq ; 32/32 ==> 32q
DIVUL.L <ea>,Dq ; 32/32 ==> 32q
```

The preferable way to divide a 64-bit dividend is:

```
DIVS.L <ea>,Dr:Dq ; 64/32 ==> 32q,32r
```

## Variants of Instruction Types

The assembler lets you 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 shown in Table 3-1.

Table 3-1. Summary of Variants of Instruction Types

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

### 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 (e.g., **#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 **ADDI** as the 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 addressing modes are necessarily legal for a particular operand of a particular instruction. The legal addressing modes for an operand vary in an irregular way, which is fully described in the publications listed in the *Motorola Compatibility* section at the beginning of this chapter.

Note that there are minor differences in legal addressing modes between the CPU32 family, 68020/30/40, and other 68000 family chips, which are described in detail in the Motorola manuals.

## Registers

The assembler recognizes the register mnemonics listed and described in Table 3-2. Register mnemonics can be uppercase or lowercase and are reserved symbols.

**Table 3-2. Summary of Register Mnemonics**

Register	Description	Supported By
A0-A7	32-Bit Address Registers.	CPU32, 68000/10/20/30/40, 68EC000/20/30/40, 68HC000/01
AC	16-Bit Access Control Register.	68851
AC0-AC1	32-Bit Access Control Register	68EC030
ACUSR	Access Control Unit Status Registers	68EC030
BAC0-BAC7	Eight 16-Bit Breakpoint Acknowledge Control Registers.	68851
BAD0-BAD7	Eight 16-Bit Breakpoint Acknowledge Data Registers.	68851

(cont.)

Table 3-2. Summary of Register Mnemonics (cont.)

Register	Description	Supported By
CAAR	Cache Control Register. Holds the address for cache control functions.	68020/30, 68EC020/30
CACR	Cache Control Register. Holds the address for cache control functions.	68020/30/40, 68EC020/30/40
CAL	8-Bit Protection Control Register.	68851
CCR	Condition Code Register. CCR is the lower eight bits of the status register (SR).	CPU32, 68000/10/20/30/40, 68EC000/20/30/40, 68HC000/01
CONTROL/ FPCR	Floating-Point Control Register.	68040, 68881/68882
CRP	64-Bit User Root Point Register.	68030, 68851
D0-D7	32-bit Data Registers.	CPU32, 68000/10/20/30/40, 68EC000/20/30/40, 68HC000/01
DFC/DFCR	Alternate Function Code Destination Register.	CPU32, 68EC010/20/30/40 68010/20/30/40
DRP	64-Bit DMA Root Point Register.	68851
DTT0-1	Transparent Translation Registers.	68040
FP0-FP7	Floating-Point Data Registers.	68040, 68881/68882
IADDR/FPIAR	Floating-Point Instruction Address Register.	68040, 68881/68882
ISP	Interrupt Stack Pointer (68020/30 interrupt state).	CPU32, 68000/10/20/30/40, 68EC000/20/30/40, 68HC000/01
ITTO-1	Transparent Translation Registers.	68040

(cont.)

Table 3-2. Summary of Register Mnemonics (cont.)

Register	Description	Supported By
MMUSR	16-Bit MMU Status Register.	68030/40, 68851
MSP	Master Stack Pointer (68020/30 supervisor state).	68020/30/40 68EC020/30/40
PC	Program Counter (used in PC-relative addressing 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 forces the use of program counter relative addressing modes.	CPU32, 68000/10/20/30/40, 68EC000/20/30/40, 68HC000/01
PCSR	16-Bit Cache Status Register.	68851
PSR	16-Bit Status Register.	68851
SCC	8-Bit Protection Control Register.	68851
SFC, SFCR	Alternate Function Code Source Register.	CPU32, 68010/20/30/40
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.	CPU32, 68000/10/20/30/40
SRP	64-Bit Supervisor Root Point Register.	68030/40, 68851
SSP/A7	System Stack Pointer.	CPU32, 68000/10/20/30/40, 68EC000/20/30/40, 68HC000/01
STATUS/FPSR	Floating-Point Status Register.	68040, 68881/68882
TC	32-Bit Translation Control Register.	68030/40, 68851

(cont.)

Table 3-2. Summary of Register Mnemonics (cont.)

Register	Description	Supported By
TT0-TT1	32-Bit Transparent Translation Registers.	68030
URP	User Root Pointer Register	68040
USP	User Stack Pointer (for user state).	CPU32, 68000/10/20/30/40, 68EC000/20/30/40, 68HC000/01
VAL	8-Bit Protection Control Register.	68851
VBR	Vector Base Register. Used for multiple vector table areas.	CPU32, 68010/20/30/40 68EC020/30/40
ZA0-ZA7	Suppressed Address Registers. The register specified is used in the instruction, but its value is considered zero for effective address calculations.	CPU32
ZD0-ZD7	Suppressed Data Registers. The register specified is used in the instruction, but its value is considered zero for effective address calculations.	CPU32
ZPC	Suppressed Program Counter. The PC is used in the instruction, but its value is considered zero for effective address calculations.	CPU32

The 68881/882 floating-point coprocessor uses the 68020/30/40 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 68020/68881/68882 processor combination provides eight Floating-Point Data Registers (**FP0** to **FP7**).

The 68881/882 interfaces to the 68020 in a way that is transparent to you. You access the floating-point registers of the 68881/882 as though they were resident in the 68020. The 68881/882 coprocessor interface has no restrictions on the use of the 68020 registers. Floating-point operations are coded the same as integer operations.

## Addressing Modes

Table 3-3 lists the addressing modes that are defined for all chips. These modes are described in more detail in the following sections.

**Table 3-3. Addressing Modes**

	68000/08/10	CPU32	68020/30/40
68302	68330	68EC020	
68EC000	68331	68EC030	
68HC000	68332	68EC040	
68HC001	68333		
	68340		
<b><u>Register Direct Modes:</u></b>			
Address Register Direct	yes	yes	yes
Data Register Direct	yes	yes	yes
<b>Memory Addressing Modes:</b>			
Address Register Indirect	yes	yes	yes
Address Register Indirect with Postincrement	yes	yes	yes
Address Register Indirect with Predecrement	yes	yes	yes
Address Register Indirect with 16-bit Displacement	yes	yes	yes
Address Register Indirect with 8-bit Displacement and Index	yes	yes	yes
Address Register Indirect with Base Displacement and Index	no	yes	yes
Memory Indirect Post-Indexed	no	no	yes
Memory Indirect Pre-Indexed	no	no	yes

(cont.)

Table 3-3. Addressing Modes (cont.)

	68000/08/10	CPU32	68020/30/40
68302	68330	68EC020	
68EC000	68331	68EC030	
68HC000	68332	68EC040	
68HC001	68333		
	68340		
<b><u>Memory Reference Modes:</u></b>			
Absolute Short Address	yes	yes	yes
Absolute Long Address	yes	yes	yes
Program Counter Indirect with 16-bit Displacement	yes	yes	yes
Program Counter Indirect with 8-bit Displacement and Index	yes	yes	yes
Program Counter Indirect with Base Displacement and Index	no	yes	yes
Program Counter Memory Indirect Post-Indexed	no	no	yes
Program Counter Memory Indirect Pre-Indexed	no	no	yes
<b><u>Non-Memory Reference Modes:</u></b>			
Immediate	yes	yes	yes

The various Program Counter relative modes all 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, i.e., the address of the first byte of extension.

The 68000/08/10 and CPU32 family microprocessors can address odd-numbered memory locations only when the instruction is operating on a single byte. Neither the assembler nor the linker checks for such odd-byte alignment and, in many cases (such as indexed addressing modes), neither the assembler nor the linker is capable of checking for this situation. The 68020/30/40 microprocessors have no such restriction. However, all chips require that instructions as opposed to data begin at

an even address, and the assembler enforces this.

### **Register Direct Modes**

The Register Direct Modes act directly on the contents of a register.

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

#### **Address Register Direct**

The Address Register Direct Mode acts directly on the contents of an address register.

#### **Data Register Direct**

The Address Register Direct Mode acts directly on the contents of a data register.

### **Memory Addressing Modes**

The Memory Addressing Modes alter memory contents.

#### **Address Register Indirect**

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

#### **Address Register Indirect with Postincrement**

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**

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 extension bytes.

#### **Address Register Indirect with 16-bit Displacement**

The Address Register Indirect with Displacement Mode calculates the memory address as the sum of the contents of an address register and a sign-extended 16-bit displacement. It requires 2 bytes of extension.

### Address Register Indirect with 8-Bit Displacement and Index

The Address Register Indirect with 8-Bit 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 8-bit displacement. It requires 2 bytes of extension.

The Index Register involved may use either all 32 bits or use 32 bits or use 16 bits sign-extended. On the CPU32 family and 68020/30/40, the Index Register contents may optionally be multiplied by a scale factor of 2, 4, or 8 before being added to the Address Register contents.

### Address Register Indirect with Base Displacement and Index

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 16- or 32-bit sign-extended base displacement. This mode requires at least 2 bytes of extension plus 2 bytes more for a 16-bit displacement or 4 bytes more for a 32-bit displacement.

The Index Register involved may use either all 32 bits or use 16 bits sign-extended. The Index Register contents may optionally be multiplied by a scale factor of 2, 4, or 8 before being added to the Address Register contents. The Address Register, Index Register and displacement may be specified as null, which would give them a value of 0. A null displacement does not require any extension bytes.

### Memory Indirect Post-Indexed

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 can be either 16 or 32 bits. This mode requires at least 2 bytes of extension plus 2 bytes more for a 16-bit displacement or 4 bytes more for a 32-bit displacement.

The Index Register involved can use either all 32 bits or use 16 bits sign-extended. The Index Register contents can optionally be multiplied by a scale factor of 2, 4, or 8 before being added to the intermediate address contents and the outer displacement. The Address Register, Index Register and displacement may be specified as null, which would give them a value of 0. A null displacement does not require any extension bytes.

## Memory Indirect Pre-Indexed

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 can be an address or a data register, and a 16- or 32-bit sign-extended base displacement. The final memory address is then calculated as the sum of the contents of the intermediate address and a 16- or 32-bit outer displacement. 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 can use either all 32 bits or 16 sign-extended bits. The Index Register contents can optionally be multiplied by a scale factor of 2, 4, or 8 before being added to the Address Register contents and the base displacement. The Address Register, Index Register and displacement may be specified as null, which would give them a value of 0. A null displacement does not require any extension bytes.

## Memory Reference Modes

The various Program Counter relative modes all 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, i.e., the address of the first byte of extension.

### Absolute Short

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). Since 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 dependent on the target chip:

1. From \$FF8000 to \$FFFFFF for the 68000/10
2. From \$F8000 to \$FFFFFF for the 68008
3. From \$FFFF8000 to \$xFFFFFFFF for the CPU32 family and 68020/30/40

Regardless of the target chip, the assembler recognizes only the absolute addresses from \$FFFF8000 to \$xFFFFFFFF as being in the high end of the 16-bit addressable memory range, also known as the short-addressable area of memory. If it is necessary to use Absolute Short Mode on the actual area of high memory that is on the target chip, any absolute code should be placed in a separate module and referenced as **XREF.S** from other modules. This technique causes the use of Absolute Short mode in most cases. Such code could also be made relocatable and placed in a **SECTION.S**, then located correctly at link time. In this case, the high address range of the code in Absolute Short mode need not be in a separate module.

### Absolute Long

Absolute Long Mode contains a full 32-bit address in the instruction and can therefore address any memory location on any chip (4 bytes of extension).

### Program Counter Indirect with 16-bit Displacement

The Program Counter Indirect with 16-bit 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.

When the target chip is the 68020, 68030, 68040, or CPU32 family, the Index Register can optionally be multiplied by a scale factor of 2, 4, or 8 before being added to the other components.

### Program Counter Indirect with 8-bit Displacement

The Program Counter 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 can 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.

When the target chip is the 68020, 68030, or 68040, or CPU32 family, the Index Register can optionally be multiplied by a scale factor of 2, 4, or 8 before being added to the other components.

### Program Counter Indirect with Base Displacement and Index

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 an address or data, and can 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 bytes more for a 16-bit displacement or 4 bytes more for a 32-bit displacement.

The Index Register may optionally be multiplied by a scale factor of 2, 4, or 8 before being added to the other components. The Address Register, Index Register, and displacement may be specified as null, which would give them a value of 0. A null displacement does not require any extension bytes.

### Program Counter Memory Indirect Post-Indexed

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 16- or 32-bit base displacement. The final memory address is then

calculated through 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 16- or 32-bit sign-extended outer displacement. 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 could use either all 32 bits or 16 bits sign-extended. The Index Register contents can optionally be multiplied by a scale factor of 2, 4, or 8 before being added to the intermediate address contents and the outer displacement. The Program Counter, Index Register, base displacement, and outer displacement can be null, which would give them a value of 0. Null displacements do not require any extension bytes.

### **Program Counter Memory Indirect Pre-Indexed**

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 can 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 can 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 can use either all 32 bits or 16 bits sign-extended. The Index Register contents can optionally be multiplied by a scale factor of 2, 4, or 8 before being added to the Program Counter contents and to the base displacement. The Program Counter, Index Register, base displacement, and outer displacement can be null, which would give them a value of 0. Null displacements do not require any extension bytes.

## **Non-Memory Reference Modes**

These modes provide data directly to the instruction.

### **Immediate**

The Immediate Mode provides data directly in the instruction. 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. 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 cannot be relocatable or external. All other immediate operands can be relocatable or external.

## 68881/882 Floating-Point and 68851 MMU Coprocessor and Addressing Modes

The 68881/882 floating-point coprocessor and 68851 MMU use 68020/30/40 addressing modes by requesting the processor to perform addressing mode calculations based on 68881/882 and 68851 instructions. The 68881/882 and 68851 know nothing about addressing modes. When instructed to do so by the 68881/882 or 68851, the 68020/30/40 evaluates the instruction, transfers the operands through the coprocessor interface, and performs the addressing mode calculations.

Any of the 68020/30/40 addressing modes described above can 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 68020/30/40 evaluates the instruction to its addressing modes. These include all legal 68020/30/40 addressing modes with the exception of a few restrictions for certain instructions. The exceptions are fully described in the *Motorola Floating-Point Coprocessor User's Manual*.

## Assembler Syntax for Effective Address Fields

The assembler uses one of several possible addressing modes depending on the operand(s) specified. The following pages describe how the assembler makes such decisions. Table 3-4 provides a definition of terms used to describe operand syntax.

Table 3-4. Assembler Syntax for Effective Address Fields

Operand	Resultant Register or Expression
A <sub>n</sub>	Represents an address register.
D <sub>n</sub>	Represents a data register.
R <sub>n</sub>	Represents either an address or data register, or a suppressed register (ZA <sub>n</sub> or ZD <sub>n</sub> ). ASM68K does not recognize the mnemonic R <sub>n</sub> .
abs_exp	Represents an absolute expression, including an external reference with no section specified.

(cont.)

Table 3-4. Assembler Syntax for Effective Address Fields (cont.)

Operand	Resultant Register or Expression
rel_exp	Represents a relocatable expression, including an external reference with a section specified.
exp	Represents either an absolute or relocatable expression.
{ }	Represents an optional field. Commas within the brackets represent a choice of elements if the field is present.

## Addressing Mode Syntax

ASM68K fully supports Motorola's 68020/30/40 syntax. This syntax uses square brackets, [ ], to designate the components of the intermediate address in the 68020/30/40 addressing modes, and parentheses to group the other components of an effective address. The following facts apply to addressing mode syntax:

- The syntax *exp(anything)* (old 68000) is equivalent to *(exp,anything)* (68020).
- The order of items separated by commas within square brackets or parentheses (grouping characters) is not significant, unless there are two A registers or two ZA 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 the Base Register and the rightmost is the Index Register.
- A 68000 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 addressing mode (perhaps with some null fields):
  1. Using a Z-register (**ZPC**, **ZAn**, or **ZDn**).
  2. Using square brackets.
  3. Specifying an explicit **L** size code on a displacement. Note that a **.W** qualifier does not force a 68020/30 mode. For example:  
`((LABEL).L,A1)`
  4. Specifying a scale factor other than 1 on an index register.
  5. Specifying a displacement too large to fit in the 68000 mode. Forward references are assumed to require 32 bits, while

externals and relocatables are assumed to require 16 bits (but if the absolute part of an expression such as *reloc+abs* is too big to fit in 16 bits, a 32-bit field will be used). 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 mode may be selected on pass 2 (some extra NOPs will trail the instruction). The OPT flags BRW and FRS do not apply to forward references which appear in conjunction with a register.

Note that coding (*exp,An*) rather than *exp(An)* or specifying a scale factor of 1 explicitly is not sufficient to force the use of 68020/30/40 modes.

Errors will occur when assembler syntax forces 68020/30/40 addressing modes, and the target microprocessor specified with the CHIP or OPT P directives is not the 68020, 68030, or 68040. You should note that:

- Assembler syntaxes that generate Address Register Indirect with Displacement or Memory Indirect modes (e.g., (*exp,An*) or ([*exp,An*],*Rn*)) allow *exp* to be an absolute or a relocatable expression. If *exp* is an absolute expression, the assembler will use it as the displacement. If *exp* is a relocatable expression, the syntax tells the assembler to access the location of the relocatable expression using register *An* indirect, and the linker will calculate the final displacement. The A2-A5 Relative Addressing section later in this chapter provides more information on this.
- Absolute expressions in operands that generate Program Counter relative addressing modes (e.g., (*abs\_exp,PC*)) can have two different meanings depending upon the assembler flag ABSPCADD.

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. For example, BRA label(PC) branches to the location of label.

When the ABSPCADD flag is off (by setting OPT NOABSPCADD or OPT -ABSPCADD), the absolute expression is the displacement. For example, BRA label(PC) branches to the address of the current PC plus the value of label.

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).

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

In the discussion that follows, the 68020/30/40 notation is used, but the list 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 relationship between operand syntax and addressing modes is described below.

### D<sub>n</sub> Operand

The operand **D<sub>n</sub>** always results in Data Register Direct Mode.

### A<sub>n</sub> Operand

The operand **A<sub>n</sub>** always results in Address Register Direct Mode.

### (A<sub>n</sub>) Operand

The operand **(A<sub>n</sub>)** always results in Address Register Indirect mode.

### (A<sub>n</sub>)+ Operand

The operand **(A<sub>n</sub>)+** always results in Address Register Indirect with Postincrement Mode.

### -(A<sub>n</sub>) Operand

The operands **-(A<sub>n</sub>)** always result in Address Register Indirect with Predecrement Mode.

### exp Operand

The operand **exp** results in Absolute Short Address, Absolute Long Address, or Program Counter Indirect with 16-bit Displacement Mode. The assembler chooses one of these modes based on rules described in this chapter. In most cases, good results will be obtained by allowing the assembler to use its default action.

### #exp Operand

The operand **#exp** results in the Immediate Mode. An absolute expression must be within a certain size range that is dependent on the instruction and qualifier code. 16- and 32-bit immediate data can be relocated but smaller fields cannot.

### **(abs\_exp,An) Operand**

The operand  $(abs\_exp,An)$  is resolved as Address Register Indirect with 16-bit Displacement Mode, provided the expression fits in 16 bits (sign-extended) and does not have an explicit .W or .L size code. An absolute external expression is considered to fit in 16 bits. The  $abs\_exp$  is used as the displacement.

If the expression does not fit in 16 bits or an explicit .W or .L is attached to it, the 68020 Address Register Indirect with Base Displacement and Index Mode is used. The specified A register is used as the Base Register, and the Index Register is null.

As a special case,  $(0,An)$  generates the more efficient Address Register Indirect Mode despite the explicit zero displacement. To generate an explicit zero displacement, you must use an external symbol.

### **(Dn), (Rn.W), (Rn.L), (abs\_exp,Dn), (abs\_exp,Rn.W), and (abs\_exp,Rn.L) Operands**

The operands  $(Dn)$ ,  $(Rn.W)$ ,  $(Rn.L)$ ,  $(abs\_exp,Dn)$ ,  $(abs\_exp,Rn.W)$  and  $(abs\_exp,Rn.L)$  generate the 68020 Address Register Indirect with Base Displacement and Index Mode. The specified register is used as the Base Register if it is an A register without a size code or scale factor attached and as the Index Register in all other cases.

### **(abs\_exp,An,Rn{.W,.L}) and (An,Rn{.W,.L}) Operands**

The operands  $(abs\_exp,An,Rn{.W,.L})$  and  $(An,Rn{.W,.L})$  result in the Address Register Indirect with 8-bit Displacement and Index Mode provided the  $abs\_exp$  fits in 8 bits sign-extended. If the  $abs\_exp$  is absent, a displacement of 0 which fits in 8 bits is used. The first A register without a size code or scale factor which is encountered, reading left to right, is the Address Register and the other register is the Index Register.

If the  $abs\_exp$  does not fit in 8 bits, the 68020 Address Register Indirect with Base Displacement and Index Mode is used. As before, the first A register without a size code or scale factor which is encountered, reading left to right, is the Base Register and the other register is the Index Register. Absolute external expressions are assumed not to fit in 8 bits.

### **Square Brackets and No PC or ZPC**

Any operand containing square brackets but not containing PC or ZPC generates one of the 68020 modes: Memory Indirect Post-Indexed or Memory Indirect Pre-Indexed. If a register (particularly, an Index Register) is specified outside the square brackets, then the Post-Indexed Mode is chosen; otherwise, Pre-Indexed Mode is chosen. Any registers and displacements not specified are null. Any relocatable

displacements are assumed to be 16 bits unless specified to be 32 bits by attaching **L**.

#### (*rel\_exp,Rn{.W,.L}*) Operand

The operand (*rel\_exp,Rn{.W,.L}*) results in Program Counter Indirect with 8-bit Displacement and Index Mode provided the *rel\_exp* is known to be in the same section as the instruction. The specified register is used as an index register. The **L** qualifier on the register indicates all 32 bits of it are to be used. The **.W** or no qualifier indicates the low order 16 bits are to be used, sign-extended. The displacement is calculated by subtracting the current PC from the *rel\_exp*; this value must fit in 8 bits sign-extended or an error occurs.

If the *rel\_exp* is not in the same section as the instruction, the Address Register Indirect with Base Displacement and Index Mode is used. The specified register is used as a Base Register if it is **An** and otherwise as the Index Register.

#### (*rel\_exp,An,Rn{.W,.L}*) Operand

The operand (*rel\_exp,An,Rn{.W,.L}*) generates the Address Register Indirect with Base Displacement and Index Mode. The first **A** register without a size code or scale factor which is encountered, reading left to right, is the Base Register and the other register is the Index Register. Relocatable displacements are assumed to require 16 bits unless specified otherwise.

The operands (*rel\_exp,Dn,Rn{.W,.L}*) and (*Dn,Rn{.W,.L}*) are invalid. One of the two registers must be an **A** register or **PC**.

#### (*exp,PC*) Operand

The operand (*exp,PC*) results in Program Counter Indirect with 16-bit Displacement Mode provided the displacement (not the actual specified *exp*) fits in 16 bits sign-extended. As noted above, when *exp* is absolute, it is by default a displacement, not an address, so in this case no further calculation needs to be done. The **OPT ABSPCADD** option overrides this default.

If a displacement needs to be calculated, the following rules are followed:

1. When the operand is in the same section as the instruction, the displacement is calculated to be the value of the *exp* minus the current value in the program counter (not the location counter).
2. When the operand is not in the same section as the instruction but is elsewhere in the current module, the value used for calculating the displacement is the offset of the operand from the beginning of its section, and the displacement is resolved finally by the linker.

3. When the operand is an external reference, the value used for calculating the displacement is 0 and the displacement is resolved finally by the linker.

If the assembly-time displacement, as determined from the rules above, does not fit in 16 bits sign-extended, the assembler chooses the Program Counter Indirect with Base Displacement and Index Mode with a null index register and 32-bit displacement field. It is possible for errors to occur at link time if the final 16-bit displacement calculated by the linker does not fit in its field.

### (*exp,PC,Rn{.W,L}*) Operand

The operand (*exp,PC,Rn{.W,L}*) results in Program Counter Indirect with 8-bit Displacement and Index Mode provided the displacement (as determined by the rules discussed above) fits in 8 bits sign-extended and does not require relocation by the linker. The latter condition means the operand must be in the same section as the instruction and cannot be external. In any other case, the operand results in the Program Counter Indirect with Base Displacement and Index Mode with a null index register. The size of the displacement field is 16 bits, unless 32 bits are to be required for the assembly time displacement or an explicit **L** is appended to the *exp*.

The operands (*exp,Dn,Rn{.W,L}*) and (*Dn,Rn{.W,L}*) are invalid. One of the two registers must be an A register or PC.

### Square Brackets and PC or ZPC

Any operand including square brackets with either PC or ZPC is resolved to one of the Program Counter Memory Indirect Post-Indexed or Program Counter Memory Indirect Pre-Indexed Mode. The PC or ZPC in either case must be inside the square brackets.

Program Counter Memory Indirect Post-Indexed Mode is chosen if there is an Index Register specified outside the square brackets; otherwise, Program Counter Memory Indirect Pre-Indexed Mode is chosen.

If the specified mnemonic were PC, the displacement would be determined as in the preceding 2 paragraphs, but if the mnemonic was ZPC, the specified *exp* for the base displacement if any is always the displacement itself, which never has the PC contents subtracted from it. Note that in the ZPC case, the PC will not be added in at runtime to create the effective address.

## Addressing Mode Selection

The assembler's rules for choosing an addressing mode for expression types are summarized in the following sections. Note that:

1. These rules do not apply to the **Bcc** or **DBcc** instructions, which always use Program Counter plus Displacement modes.
2. The final choice between addressing modes Absolute Short Address and Absolute Long Address is determined 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 the Program Counter Indirect with 16-bit Displacement Mode, nor will they cause a reference to a location that is known to be in short-addressable memory to use Absolute Long mode.

## ABS Section

The rules for determining an addressing mode for expression types for an ABS instruction section are listed in the following subsections.

### **abs\_exp**

If **OPT P** is set and the displacement is within a 16-bit range, then the Program Counter Indirect with 16-bit Displacement Mode is used.

If **OPT P** is not set or the displacement is not within a 16-bit range, then Absolute Short Address Mode is used if the operand is in short-addressable memory; otherwise, Absolute Long Address Mode is used.

### External Reference in Specified Section

If the section of the operand is short, then Absolute Short Address Mode is used; otherwise, Absolute Long Address Mode is used.

### External Reference in Unspecified Section

If the operand was defined in **XREF.S** or if **OPT F** is set, then Absolute Short Address Mode is used; otherwise, Absolute Long Address Mode is used.

### **rel\_exp**

If the section of the operand is short, then Absolute Short Address Mode is used; otherwise, Absolute Long Address Mode is used.

### **unknown forward ref**

If **OPT F** is set, then two bytes are allocated; otherwise, four bytes are allocated.

## REL Section

The rules for determining an addressing mode for expression types for an **REL** instruction section are listed in the following subsections.

### **abs\_exp**

If the operand is in short-addressable memory, then Absolute Short Address Mode is used; otherwise, Absolute Long Address Mode is used.

### External Reference in Specified Section

If **OPT R** is set then the Program Counter Indirect with 16-bit Displacement Mode is used; otherwise, the Absolute Short Address Mode is used if the section of the operand is short and the Absolute Long Address Mode is used if it is not.

### External Reference in Unspecified Section

If the operand was defined in **XREF.S** or if **OPT F** is set, then Absolute Short Address Mode is used; otherwise, Absolute Long Address Mode is used.

### **rel\_exp**

If **OPT R** is set, then the Absolute Long Address Mode is used.

If **OPT R** is not set, the operand and instruction are in the same section, and the displacement is within a 16-bit range, then the Program Counter Indirect with 16-bit Displacement Mode is used; otherwise, Absolute Short Address Mode is used if the operand is in short-addressable memory and Absolute Long Address Mode is used if it is not.

### **unknown forward ref**

If **OPT F** is set, then two bytes are allocated; otherwise, four bytes are allocated.

## User Control of Addressing Modes

The default addressing mode using ASM68K is Absolute Long Mode (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 will want to choose a more compact and faster mode in some cases.

The choice of mode can be controlled in several ways:

1. Relocatable sections or external references can be specified as short (see Chapter 5, *Relocation*, for further information), meaning that any references

to those sections and external references will use Absolute Short Mode in preference to Absolute Long Mode (but not in preference to other modes). Short sections and external references are always placed in the short-addressable areas of memory by the linker.

2. The option flag **P** can be set with the **OPT** directive, causing all references to a known absolute location from an absolute location to use Program Counter Indirect with 16-bit Displacement Mode provided the displacement is within 16-bit range.
3. The option flag **R** can be set with 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 on the **XREF** directive) to use Program Counter Indirect with 16-bit Displacement mode. Most such references must be resolved by the linker.

This option requires careful use because those locations that are not within 16-bit displacement from the current PC will cause errors, either in the assembler or in the linker. These errors will not happen if the default Absolute Long Mode is used.

4. The option flag **F** can be set with 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 requires careful use 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 addressing mode can be selected in which case the final 2 bytes will be filled with a NOP.

5. The option flag **B** can be set with 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. Possibly, there may not be room for the actual displacement, and unnecessary errors could occur.
6. Individual **Bcc**, **JMP**, and **JSR** instructions can apply the **.S** or **.L** qualifiers to the opcode in order to force the 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 addressing 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 cannot 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** cannot reference an external reference or any location outside the instruction section since the linker cannot resolve 8-bit displacements.

## A2-A5 Relative Addressing

Memory locations are commonly accessed relative to the program counter or via absolute addresses. 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 command.

ASM68K does not restrict the use of relative addressing with A0, A1, A6, and A7. Rather it supports the compiler run-time requirements for **An**-relative data access.

A2-A5 relative addressing is useful when:

- Accessing statically allocated data areas. This 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.

## 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. If possible, the displacements are calculated by the assembler when *exp* is an absolute expression or by the linker when *exp* is a relocatable expression.

## Absolute Expressions versus 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 to 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 will calculate the displacements with the following equations:

$$\begin{aligned} \text{effective\_address} &= A_n + \text{displacement} \\ \text{displacement} &= \text{effective\_address} - A_n \\ \text{displacement} &= \text{relocatable\_expression} - A_n \end{aligned}$$

The linker knows the value of the relocatable expression; however, it does not know what will be in  $A_n$  when the expression executes.

To solve the linker's problem of not knowing the run-time contents of an address register and to let you use relocatable expressions in conjunction with the powerful address register indirect with displacement modes, the linker **INDEX** command was created to let you specify the run-time value of  $A_n$ .

### The INDEX Linker Command

The **INDEX** command lets you 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 ? $A_n$  (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 considers the run-time value of  $A_n$  to be zero.

### Accessing Statically Allocated Areas

The 68000 address register indirect with displacement addressing modes (for example, those modes generated for syntaxes such as  $\exp(A_n)$  or  $(\exp,(A_n,R_n)$ , 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 as shown in Figure 3-1. Notice that the address register indirect mode is coded in two fewer bytes than the absolute long mode.

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

```
Microtec Research ASM68K Version x.y      Wed Jul 22 13:23:44 1992    Page 1

Command line: /usr4/engr.sun4/bin/asm68k -l TEST1
Line Address
1          SECT   DATA
2 00000000     WORD1 DS.W  1
3 00000002     DS.B   0FFFEH
4                                     ; Address Mode Generated
5          SECT   CODE   ;
6 00000000 3039 0000 0000 R  MOVE  WORD1,D0  ; Absolute Long
7 00000005 302A 0000       R  MOVE  WORD1(A2),D0  ; Address Reg. Indirect
8                                     ; with Displacement
9          END

Microtec Research ASM68K Version x.y      Wed Jul 22 13:23:44 1992    Page 2

Symbol Table
Label      Value
WORD1     DATA:00000000
```

Figure 3-1. Absolute versus Indirect Addressing Modes

## Accessing Dynamically Allocated Areas

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

## Example Listings

The following listings provide examples of A2-A5 relative addressing and how to use the **INDEX** command. The assembled source file output listing is shown in Figure 3-2. The linker map file in Figure 3-3 shows the **INDEX** command used with an offset. The linker map file in Figure 3-4 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.

**Example 1:**

```
Microtec Research ASM68K Version x.y Wed Jul 22 13:11:53 1992 Page 1

Command line: /usr4/engr.sun4/bin/asm68k -l TEST2
Line Address
1
2          XREF    ?A2      ; This symbol is defined by
3          ; the linker INDEX command.
4
5          XDEF    VAR      ; To get the effective address
6          ; on the linker listing.
7
8          SECT    DATA
9 00000000          DS.B    6000H
10 00006000         VAR    DS.B  9FFFH ; Effective address of VAR =
11          ; load address of DATA section
12          ; + 6000H.
13
14          SECT    PROG
15 00000000 247C 0000 0000 E  MOVE.L #?A2,A2 ; Initialize A2 with run-time
16          ; value specified in the INDEX
17          ; command.
18
19 00000006 426A 6000     R    CLR    VAR(A2) ; Address Register Indirect
20          ; with Displacement mode
21          ; is generated. When this
22          ; module is linked, linker
23          ; will calculate the 16-bit
24          ; displacement of A2 (as
25          ; specified by the I6 (NDEX
26          ; command) from the
27          ; effective address of VAR.
28          END
```

**Figure 3-2. A2-A5 Relative Addressing Example**

Symbol Table	
Label	Value
?A2	External
VAR	DATA:00006000

Figure 3-2. A2-A5 Relative Addressing Example (cont.)

**Example 2:**

```
Microtec Research LNK68K Version x.y   Wed Jul 22 13:42:08 1992   Page  1
Command line: /usr4/engr.sun4/bin/lnk68k -mc TEST3.opt

LIST C ; Include a cross reference
; table on the output listing.

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=0FF8000H ; Run-time value of A2 = 0FF8000H + 8000H,
; = 0FF8000H.

* The displacement calculated for the "CLR VAR(A2)" instruction
* is the effective address of VAR (0FF8000 + 6000H) minus the
* run-time value of A2 (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 TEST2
END
```

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

Microtec Research LNK6BK Version x.y    Wed Jul 22 13:48:45 1992    Page 2																							
OUTPUT MODULE NAME: TEST3 OUTPUT MODULE FORMAT: IEEE																							
<u>SECTION SUMMARY</u>																							
<table><thead><tr><th>SECTION</th><th>ATTRIBUTE</th><th>START</th><th>END</th><th>LENGTH</th><th>ALIGN</th></tr></thead><tbody><tr><td>PROG</td><td>NORMAL CODE</td><td>00001000</td><td>00001009</td><td>0000000A</td><td>2 (WORD)</td></tr><tr><td>DATA</td><td>NORMAL DATA</td><td>00FF0000</td><td>00FFFFFE</td><td>0000FFFF</td><td>2 (WORD)</td></tr></tbody></table>						SECTION	ATTRIBUTE	START	END	LENGTH	ALIGN	PROG	NORMAL CODE	00001000	00001009	0000000A	2 (WORD)	DATA	NORMAL DATA	00FF0000	00FFFFFE	0000FFFF	2 (WORD)
SECTION	ATTRIBUTE	START	END	LENGTH	ALIGN																		
PROG	NORMAL CODE	00001000	00001009	0000000A	2 (WORD)																		
DATA	NORMAL DATA	00FF0000	00FFFFFE	0000FFFF	2 (WORD)																		
<u>MODULE SUMMARY</u>																							
<table><thead><tr><th>MODULE</th><th>SECTION:START</th><th>SECTION:END</th><th>FILE</th></tr></thead><tbody><tr><td>TEST2 EST2.0</td><td>DATA:00FF0000 PROG:00001000</td><td>DATA:00FFFFFE PROG:00001009</td><td>/test/TEST2.0</td></tr></tbody></table>						MODULE	SECTION:START	SECTION:END	FILE	TEST2 EST2.0	DATA:00FF0000 PROG:00001000	DATA:00FFFFFE PROG:00001009	/test/TEST2.0										
MODULE	SECTION:START	SECTION:END	FILE																				
TEST2 EST2.0	DATA:00FF0000 PROG:00001000	DATA:00FFFFFE PROG:00001009	/test/TEST2.0																				
<u>CROSS REFERENCE TABLE</u>																							
<table><thead><tr><th>SYMBOL</th><th>SECTION</th><th>ADDRESS</th><th>MODULE</th></tr></thead><tbody><tr><td>VAR</td><td>DATA</td><td>00FF6000</td><td>-TEST2</td></tr><tr><td>?A2</td><td></td><td>00FF0000</td><td>-\$\$ TEST2</td></tr></tbody></table>						SYMBOL	SECTION	ADDRESS	MODULE	VAR	DATA	00FF6000	-TEST2	?A2		00FF0000	-\$\$ TEST2						
SYMBOL	SECTION	ADDRESS	MODULE																				
VAR	DATA	00FF6000	-TEST2																				
?A2		00FF0000	-\$\$ TEST2																				
START ADDRESS: 00000000																							
Load Completed																							

Figure 3-3. Using the INDEX Command with Offset (cont.)

**Example 3:**

```
Microtec Research LNK68K Version x.y   Wed Jul 22 13:48:42 1992  Page  1
Command line: /usr4/engr.sum4/bin/lnk68k -mc TEST4.opt

LIST C ; Include a cross reference table on the output
; listing.

INDEX ?A2,DATA,0 ; The run-time value of A2 equals the
; load address of the DATA section.

SECT DATA=0FF0000H ; Run-time value of A2 = 0FF0000H.

* The displacement calculated for the "CLR VAR(A2)"
* instruction is the effective address of VAR (0FF0000 +
* 6000H) minus the run-time value of A2 (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 TEST2
END
```

**Figure 3-4. Using the INDEX Command without Offset**

```
Microtec Research LNK6BK Version x.y      Wed Jul 22 13:48:45 1992      Page 2

OUTPUT MODULE NAME: TEST4
OUTPUT MODULE FORMAT: IEEE

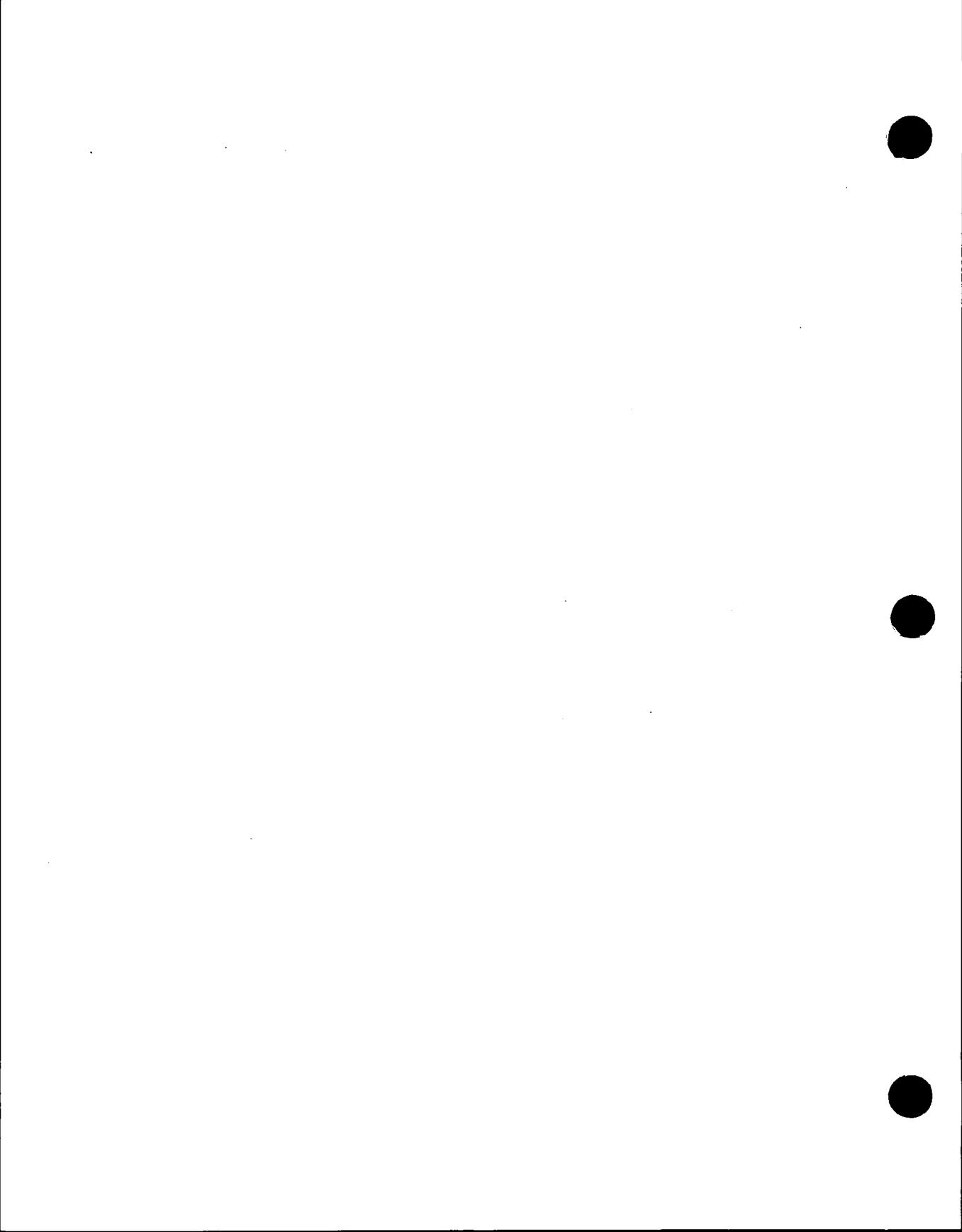
SECTION SUMMARY
-----
SECTION ATTRIBUTE                      START     END      LENGTH   ALIGN
PROG    NORMAL CODE                   00001000  00001009  0000008A  2 (WORD)
DATA    NORMAL DATA                  00FF0000  00FFFFFF  0000FFFF  2 (WORD)

MODULE SUMMARY
-----
MODULE      SECTION:START          SECTION:END      FILE
TEST2        DATA:00FF0000          DATA:00FFFFFF /test/TEST2.8
EST2.o       PROG:00001000          PROG:00001009

CROSS REFERENCE TABLE
-----
SYMBOL                         SECTION      ADDRESS      MODULE
VAR                           DATA         00FF6000  -TEST2
?A2                           DATA         00FF8000  -$$
                                MODULE      TEST2

START ADDRESS: 00000000
Load Completed
```

Figure 3-4. Using the INDEX Command without Offset (cont.)



## Introduction

Object modules produced by ASM68K are in a relocatable format which lets you write modular programs whose final addresses will be adjusted by the LNK68K Linker. Individual program modules can be changed without reassembling the entire program, and 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 final link time.
- Large programs can be easily separated into smaller modules, developed separately, and linked together.
- If one module contains an error, only that module needs to be modified and reassembled.
- Once developed, a library of routines can be used by many people.
- 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 logically distinct from other sections. At link time, the addresses for each section can be specified separately.

Sections are identified by names that follow the syntactic rules for symbols. Section names can duplicate labels or register names without conflict; they can be any symbol or a two-digit decimal number. Section names can appear in COMMON, SECT (or SECTION), and XREF directives as well as in the .SIZEOF. and .STARTOF. operators.

ASM68K provides for up to 200 program sections including both numbered and named sections. LNK68K can link up to 32,767 sections. One of these is predefined to be 00 (numbered section 0). Each relocatable section has four attributes:

- Common/noncommon
- Short/long

- Section type
- Section alignment

## Common versus 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 linker 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.

You should avoid putting instructions or code-generating directives (**DC**, **DCB**) in common sections. If you initialize the same common section in two different modules, both sets of code will be loaded into the same memory locations by the linker, which does not report this occurrence as an error. This can obviously cause problems. On the other hand, initializing a common section in only one module can be useful. The assembler gives a **W** flag (warning) whenever it sees bytes generated in a common section.

In a given assembly, a section name can 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.

The common or noncommon attribute can 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 linker can do so. In this instance, the section name must appear in a **SECT** or **COMMON** directive in another assembly; one whose object module is included in the link.

For more information on these attributes, refer to *Relocatable Sections* in Chapter 9, *Linker Operation*.

## Short versus 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 linker will load all short sections into the areas of memory addressable with 16-bit absolute addresses.

In certain situations, the assembler will choose a more compact addressing mode

when a reference is made to a short section (see Chapter 3, *Instructions and Addressing Modes*, for more information). Otherwise, short and long sections are treated alike.

For more information on these attributes, refer to *Relocatable Sections* in Chapter 9, *Linker Operation*.

## Section Alignment Attribute

The section alignment attribute affects the beginning address of each file's contribution to a section. 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 the specified alignment boundary.

The section alignment attribute may be either 1, 2 or 4. The default alignment of a section is 2 if the linker CHIP command does not specify a 32-bit target processor such as the 68020, 68030, 68040, 680330, 680331, 680332, 680333, 680340, 68EC020, 68EC030, 68EC040, or CPU32.

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

The alignment attribute is specified in the **SECTION** assembler directive as shown in the following example:

```
SECTION A, 4
```

If you have specified the alignment attribute differently in several files, the following rules apply:

- If you have specified an **ALIGN** linker command for a section, all relocatable subsections of that section are aligned modulo the greater of the two alignments.
- If there is no **ALIGN** linker command, each file's contribution to a section can have a different alignment attribute as specified in the file. However, if an alignment attribute of 4 is specified anywhere for a section, the first file which contributes to the section is aligned modulo 4, taking precedence over any alignment attributes in that first file.

For more information on these attributes, refer to *Relocatable Sections* in Chapter 9, *Linker Operation*.

## Section Type Attributes

There are four types of relocatable sections:

1. C — Program Code
2. D — Data
3. M — Mixed code and data
4. R — ROMable Data

The SECTION assembler directive lets you explicitly specify a section's type by adding a C, D, M, or R qualifier to the SECTION directive as described in Chapter 5, *Assembler Directives*.

The section type attribute serves as documentation to remind you of what a section contains. Program code sections generally contain instructions. Data sections generally contain read/write data items. ROMable data sections generally contain read-only data items.

The section type attribute affects the production of 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.

You can specify the section type attribute explicitly in the SECTION directive.

### Example:

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

If you do not explicitly specify the section type attribute, the assembler assigns the section type according to the following rules after encountering the SECTION directive:

1. If the assembler encounters instructions only, the assembler will set the section type attribute to program code (C).
2. If the assembler encounters a data definition directives only, the assembler will set the section type attribute to data (D).
3. If both instructions and data definition directions are encountered, the assembler will set the section type attribute to mixed (M).

For more information on these attributes, refer to *Relocatable Sections* in Chapter 9, *Linker Operation*.

## Section Types and HP 64000 Symbolic Files

The HP 64000 symbolic files `asmb_sym` and `link_sym` supply program symbol information to HP 64000 emulators and analysis tools. For more information on these files, see Appendix A, *HP 64000 Development System Support*, in your *ASM68K User's Guide*.

When producing HP 64000 symbol files, the symbols from the various sections are mapped onto the HP 64000 sections as shown in Table 4-1.

Table 4-1. ASM68K/HP 64000 Section Mapping

ASM68K Section	HP 64000 Section
C (program code)	PROG
D (data)	DATA
R (ROMable data)	COMN
Extra (code, data, and ROMable data)	ABS
ORG (absolute)	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, one section maximum can be mapped to **PROG**, one section to **DATA**, and one 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 shown in Table 4-2, this mapping conflicts with the HP 64000 file formats. The assembler and linker then do the following:

1. The second and subsequent sections that map to either **PROG**, **DATA**, or **COMN** are called **extra CODE**, **DATA**, and **ROM** sections

2. The symbols from **extra** sections are omitted from the HP 64000 assembler symbol file, which means that local symbols from **extra** sections will not be available at assembler analysis time. When this happens, the assembler issues the following warning:

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

3. The code from the **extra** section is correct and is treated normally
4. The linker, when producing a **link\_sym** file, maps the symbols from **extra** sections to HP 64000 ABS sections. The symbol values are correct. They appear as ABS on HP emulators and analysis tools.

Because the assembler allows many relocatable sections, it is sometimes impossible to produce perfect HP 64000 assembler symbol and linker symbol files. In these situations, the code is 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 which lets you place the sections in a RAM/ROM environment. Common sections are generally used for program variables that reside in RAM. Common sections are similar to named **COMMON** in FORTRAN. As with nonrelocatable assemblers, you can 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 4-2 illustrates how a section is assigned the common/noncommon and short/long attributes. An example of how to use this table follows:

The first time a section name appears, it has no previous attributes; therefore, the first horizontal row of the table is marked undefined. 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.

**Table 4-2. How the Assembler Assigns Section Attributes**

Previous Section Attribute	New Statement in Which the Section Name Appears					
	XREF	XREF.S	SECT	SECT.S	COMMON	COMMON.S
UNDEFINED	XREF-ONLY LONG	XREF-ONLY SHORT	NONCOMMON LONG	NONCOMMON SHORT	COMMON LONG	COMMON SHORT
XREF-ONLY LONG	XREF-ONLY LONG	XREF-ONLY* SHORT	NONCOMMON LONG	NONCOMMON SHORT	COMMON LONG	COMMON SHORT
XREF-ONLY SHORT	XREF-ONLY* SHORT	XREF-ONLY SHORT	NONCOMMON* SHORT	NONCOMMON 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
NONCOMMON LONG	NONCOMMON LONG	NONCOMMON* SHORT	NONCOMMON LONG	NONCOMMON* SHORT	ERROR	ERROR
NONCOMMON SHORT	NONCOMMON* SHORT	NONCOMMON SHORT	NONCOMMON* SHORT	NONCOMMON SHORT	ERROR	ERROR

\* A warning message is produced

## Linking

The object modules produced by the assembler are combined or linked together by a linker. The linker 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 by the linker. External references are symbols referenced in one module but defined in another module. The linker combines the external definitions from one program with the external references from other programs to obtain the final addresses. A program can contain both external references and definitions.

## Relocatable versus Absolute Symbols

Each symbol in the assembler has an associated 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 can both appear in an absolute or in a relocatable program section.

Absolute symbols are defined as follows:

- Any 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 addressing modes.

Relocatable symbols are defined as follows:

- Any 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 with a section attached 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. Expressions containing undefined or external symbols are relocatable. All numeric constants are considered absolute. Relocatable expressions can 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-ABS REL+REL REL-REL  
REL\*ABS ABS\*REL REL/ABS ABS/REL*

In addition, the following valid expressions 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* denotes a relocatable symbol or expression.

*REL=REL*  
*REL<REL*  
*REL-REL*

*REL>REL*  
*REL>=REL*  
*REL+REL*

*REL<=REL*  
*REL>REL*

## Complex Expressions

A complex relocatable expression results when two relocatable expressions are subtracted or added together, or when a relocatable expression is subtracted from an absolute expression. Only the plus (+), minus (-), multiplication (\*), and division (/) operators are allowed within these subexpressions. Results from using the (/) operator are truncated to an integer value (i.e.,  $5 / 2 = 2$ ).

Complex relocatable expressions are not legal for use with the **ORG**, **OFFSET**, **COMLINE**, **END**, **FAIL**, **SPC**, and **LLEN** directives.

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.
- 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 a single, negated start address of a relocatable section or references to the start addresses of two or more relocatable sections.

### Note

For information on valid expressions, see the section *Expressions* in Chapter 2, *Assembly Language*.



# Assembler Directives 5

## Introduction

Assembler directives are written as ordinary statements in the assembler language. Rather than being translated into equivalent machine language, they are interpreted as commands to the assembler. With these directives, the assembler reserves memory space, defines bytes of data, assigns values to symbols, controls the output listing, etc.

This chapter describes all directives (also called Pseudo-Ops) except those primarily associated with macro assembly, structured syntax, and array relative addressing, which are described in later chapters.

## Assembler Directives

The notational conventions used to describe the syntax of assembler directives are the same as that presented in the preface of this manual. The assembler directives in this chapter are organized alphabetically. An alphabetical listing of the assembler directives is shown in Table 5-1.

Table 5-1. Alphabetical listing of the Assembler Directives

Directive	Function
ALIGN	Specifies Instruction Alignment
CHIP	Specifies Target Microprocessor
COMLINE	Defines Storage
COMMON	Specifies Common Section
DC	Defines Constant Value
DCB	Defines Constant Block
DS	Defines Storage
ELSEC	Conditional Assembly Converse
END	End of Assembly

(cont.)

**Table 5-1. Alphabetical listing of the Assembler Directives (cont.)**

<b>Directive</b>	<b>Function</b>
ENDC	Ends Conditional Assembly
ENDR	Ends Repeat
EQU	Equates a Symbol to an Expression
FAIL	Generates Programmed Error
FEQU	Equates a Symbol to a Floating Expression
FOPT	Specifies Floating-Point Options for Assembly
FORMAT	Formats Listing
IDNT	Specifies Module Name
IFC	Checks if Strings Equal (Conditional Assembly)
IFDEF	Checks if Symbol Defined (Conditional Assembly)
IFEQ	Checks if Value Equal to Zero (Conditional Assembly)
IFGE	Checks if Value Non-Negative (Conditional Assembly)
IFGT	Checks if Value Greater Than Zero (Conditional Assembly)
IFLE	Checks if Value Non-Positive (Conditional Assembly)
IFLT	Checks if Value Less Than Zero (Conditional Assembly)
IFNC	Checks if Strings Not Equal (Conditional Assembly)
IFNDEF	Checks if Symbol Not Defined (Conditional Assembly)
IFNE	Conditional Assembly Unequal to Zero Test
INCLUDE	Includes Source File

(cont.)

**Table 5-1. Alphabetical listing of the Assembler Directives (cont.)**

<b>Directive</b>	<b>Function</b>
IRP	Specifies Indefinite Repeat
IRPC	Specifies Indefinite Repeat Character
LIST	Generates Assembly Listing
LLEN	Sets Length of Line in Assembler Listing
MASK2	Assembles for R9M Chip
NAME	Specifies Module Name
NOOBJ	Does Not Create an Output Object Module
OFFSET	Defines Table of Offsets
OPT	Sets Options for Assembly
ORG	Begins an Absolute Section
PAGE	Advances Listing Form to Next Page
PLEN	Sets Length of Listing Page
REG	Defines a Register List
REPT	Specifies Repeat
RESTORE	Restores Options
SAVE	Saves Options
SECT / SECTION	Specifies Section
SET	Equates a Symbol to an Expression
SPC	Spaces Lines on Listing
TTL	Sets Program Heading
XCOM	Specifies Weak External Reference
XDEF	Specifies External Definition
XREF	Specifies External Reference

## ALIGN — Specifies Instruction Alignment

### Syntax

```
ALIGN {1 | 2 | 4}
```

### Description

- |   |                               |
|---|-------------------------------|
| 1 | Byte alignment (8 bits).      |
| 2 | Word alignment (16 bits).     |
| 4 | Longword alignment (32 bits). |

The ALIGN directive specifies the modulo number of bytes to which the address of the next instruction is to be aligned.

### Notes

It is not recommended that instructions be put in sections with byte alignment because you cannot guarantee that the starting address of an instruction will end up on an even boundary during link time. If the starting address does not end up on an even boundary, you will have an illegal address.

### Example

```
ALIGN 2
```

In the example, the address is aligned to word alignment.

## CHIP — Specifies Target Microprocessor

### Syntax

```
CHIP type[/cotype]
```

### Description

<i>type</i>	Specifies target processor. Valid values are: 68000, 68008, 68010, CPU32, 68020, 68030, 68040, 68HC000, 68HC001, 68EC000, 68330, 68331, 68332, 68333, 68340, 68EC020, 68EC030, or 68EC040. (default: 68000)
<i>cotype</i>	Specifies target coprocessor. Valid values are 68881 and 68851. When using the 68881, 68040 processor type is illegal. When using the 68851, the 68030 and 68040 processor types are illegal. (default: 68881)

The **CHIP** directive specifies the target microprocessor. Table 5-2 shows target processor values and the processor instruction set supported.

Table 5-2. Processor Identification

Processor Value	Instruction Set Supported
68000	68000
68008	68000
68302	68000
68EC000	68000
68HC000	68000
68HC001	68000
68010	68010
68020	68020
68EC020	68020
68030	68030
68EC030	68EC030

(cont.)

Table 5-2. Processor Identification (cont.)

Processor Value	Instruction Set Supported
68040	68040
68EC040	68EC040
68330	CPU32
68331	CPU32
68332	CPU32
68333	CPU32
68340	CPU32
CPU32	CPU32

The differences from the assembler's point of view are as follows:

1. Beyond standard 68000 family instructions, the 68010 has the additional instructions **MOVEC**, **MOVES**, **RTD** and **MOVE** from CCR. If one of these instructions is encountered when the **CHIP** is not set to 68010, a warning is generated and the 68010 instruction is assembled.
2. The 68020 carries the standard instruction set of the 68000 family and, unlike the 68010, has the additional instructions **BFCHG**, **BFCLR**, **BFEEXTS**, **BFEEXTU**, **BFFF0**, **BFINS**, **BFSET**, **BFTST**, **BKPT**, **CALLM**, **CAS**, **CAS2**, **CHK2**, **CMP2**, **DIVSL**, **DIVUL**, **PACK**, **RTM**, **TDIVS**, **TDIVU**, **TRAPcc**, **Tcc**, **TPcc**, and **UNPK**. It has six new addressing modes as described in the 68020 instructions **DIVS**, **DIVU**, **EXTB**, **LINK**, **MOVEC**, **MULS**, **MULU**, and **TST**. Using any of these constructs when the **CHIP** is not set to 68020, 68030, or 68040 produces an error.

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

3. The 68030 has the standard 68000 family instructions, as well as the additional instructions **PFLUSH**, **PFLUSHA**, **PLOADR**, **PLOADW**,

**PMOVE, PMOVEFD, PTESTR, and PTESTW.** It also has the additional registers **CRP, SRP, TC, TT0 to TT1, and MMUSR.**

4. The 68851 has the standard 68000 family instructions, as well as the additional instructions **PBcc, PDBcc, PFLUSH, PFLUSHA, PFLUSHHS, PFLUSHR, PLOADR, PLOADW, PMOVE, PRESTORE, PSAVE, PScc, PTESTR, PTESTW, PTRAPcc, and PVALID.** It also has the additional registers **VAL, CAL, SCC, CRP, SRP, DRP, TC, AC, PCSR, PSR, BAD0 to BAD7, and BAC0 to BAC7.**
5. The CPU32 family has the same instructions as the 68010 with the following additions: **Bcc.L, BGND, BRA.L, BSR.L, CHKL, CHK2, CMP2, DIVS.L, DIVSL, DIVUL, DIVU, EXTB, LINK.L, LISTOP, MULS.L, MULUL, TBLS, TBLSN, TBLU, TBLUN, TRAPcc, TST An, and TST #<data>.**
6. The 68040 has the same instructions set as the 68020 and 68030 with the following additions: **CINVL, CINVP, CINVA, CPUSHA, CPUSHL, CPUSHP, MOVE16, PFLUSHAN, and PFLUSHN.**

If no **CHIP** or **OPT P** (which has the same function) directive appears, the default target processor is **68000**.

### Example

CHIP 68020

In the example above, the 68020 processor is specified.

## COMLINE — Specifies Memory Space

### Syntax

COMLINE *n*

### Description

*n*                   Specifies the number of bytes in memory to be reserved.

The COMLINE directive in the source code reserves a block of sequential memory locations (in bytes). The argument is the specified number of bytes (e.g., COMLINE 8 reserves 8 bytes in memory).

COMLINE is supplied for Motorola compatibility. ASM68K treats COMLINE as a DS.B instruction.

## COMMON — Specifies Common Section

### Syntax

```
[label] COMMON[.S] {sname | snumber}[,n][,type][,hptype]]
```

### Description

<i>label</i>	Specifies the section name. If <i>sname</i> has been specified, <i>label</i> will be assigned the address of the current location counter (it is a normal label).
.S	Specifies whether the section has the short attribute.
<i>sname</i>	The section name. Any valid section can be used.
<i>snumber</i>	The section number. One or two decimal digits can be used. If <i>label</i> is not specified, <i>snumber</i> is treated as the name of the common section.  If <i>label</i> is specified, <i>label</i> is appended to <i>snumber</i> to create the name of a unique common section, such as <b>00label</b> . This common section is not considered to have any particular connection with the noncommon (or common) numbered section 0.
<i>n</i>	The number of bytes of alignment. It can be 2 or 4. (default = 2)
<i>type</i>	The type of section. It can be:  D           Data R           ROMable Data
<i>hptype</i>	The HP 64000 section type. It can be:  A           ABS D           DATA C           COMN

The **COMMON** directive causes the statements following it to be assembled in relocatable mode using the named common section. 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.

You can alternate between various sections within one program by using multiple **SECT** and **COMMON** directives. The assembler will maintain the current value of the location counter for each section.

No executable code should be placed in a common section. Therefore, any directive that generates code (DC or DCB) and any instruction will be flagged with a warning when used in a common section. Typically, the DS directive will be used to allocate storage within 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.

The *hptype* refers to the HP 64000 section type. If it is not specified, ASM68K assigns a *hptype* field according to the following rule: data sections map to DATA and ROMable sections map to COMN subject with the following restrictions:

1. There can be at most one DATA and one COMN section for each module.
2. The second and subsequent sections assigned to DATA or COMN are called extra sections. Extra sections have ABS in their *hptype* field, and their local symbols cannot be written to the HP asmb\_sym file.

You cannot override the first restriction by explicitly assigning more than one section to DATA or COMN. If you do so, ASM68K will issue the following warning:

Maximum number of typed sections exceeded in HP mode

This warning is issued whenever an extra section is created regardless of whether the *hptype* was set explicitly or implicitly.

If *hptype* is ABS, the section must be treated as an extra section; that is, the section will have no symbols in the asmb\_sym file because it is impossible to relocate them.

### Example

```
LABEL1 COMMON  SECT1  ; Name is SECT1, LABEL1 is
                  ; normal symbol
      COMMON  CODE    ; Name is CODE
      COMMON  1,4,D   ; Name is 1, common section
                  ; quad aligned, containing data
LABEL1 COMMON  1     ; Name is 1LABEL1, common section
                  ; No conflict with other LABEL1
```

## DC — Defines Constant

### Syntax

[*label*] DC[.qualifier] *operand*[,*operand*] ...

### Description

*label*      Specifies an optional label that will be assigned the address of the first byte defined.

*qualifier*      Specifies an optional qualifier that can be:

- .B      Byte data
- .W      Word data
- .L      Longword data
- .S      Single precision floating
- .D      Double precision floating
- .X      Extended floating
- .P      Packed binary coded decimal

The default is .W.

*operand*      Specifies a character string or an expression for qualifiers .B, .W, and .L. All expressions are calculated as 32-bit values. For .B, this value must fit in 8 bits (either zero-filled or one-filled); for .W, it must fit in 16 bits. If this condition is violated, an error is produced.

Specifies a floating-point number for qualifiers .S, .D, .X, and .P. If this number cannot be stored in the indicated number of bits (because its exponent is too large), an error is reported. However, excessive bits of precision in a specified mantissa are truncated without a warning.

The DC directive defines up to 509 bytes of data. The assembly program counter symbol is represented by an asterisk (\*). This symbol gets evaluated to be the address of the beginning of the DC command plus the number of operands preceding the asterisk. Motorola evaluates it at the beginning of the DC command.

Operands of a DC.W or DC.L can be relocatable while operands of a DC.B are not relocatable. Operands of a DC.S, DC.D, DC.X, and DC.P can 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 zeros 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.

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**, and eight bytes per operand for a **DC.D**. The operand must evaluate to a value that fits into the specified number of bytes or an error is generated. Negative values are stored using their two's complement representation.

The **.S** and **.D** qualifiers permit definition of single- and double-precision floating-point numbers, respectively. The generated bit patterns are IEEE standard and are compatible with the Motorola MC68881/MC68882 chip. 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.

Floating-point numbers can 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.

### Examples

```
3.14159  
-22E-100; Equivalent to -22 * 10-100
```

Underscores can 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. You are 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.X**, **DC.P**, and **DC.D** directives will accept only floating-point numbers as operands. **DC** with any other qualifier will not accept floating-point numbers as operands.

**Example (with generated bytes shown)****Generated Bytes    Directive    Operand**

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		

## DCB — Defines Constant Block

### Syntax

[*label*] DCB[.*qualifier*] *length,value*

### Description

*label*      Specifies a label that will be assigned the address of the first byte allocated.

*qualifier*      Defines the units in which storage is allocated. The units can be any of the following:

- .B      Bytes
- .W      Words
- .L      Longwords
- .S      Single precision
- .D      Double precision
- .X      Extended floating
- .P      Packed binary coded decimal

The default is .W.

*length*      Defines the number of units of storage to allocate. This absolute expression cannot contain forward, undefined, or external references.

*value*      Defines the initial value for each unit. For qualifiers .B, .W, and .L, this is an expression that can contain forward references, relocatables, externals, or complex expressions. For qualifiers .S, .D, .X, and .P, this is a floating-point number as described under the DC directive.

The DCB directive causes the assembler to allocate a block of bytes, words, longwords, single-precision floating-point numbers (32 bits), double-precision floating-point numbers (64 bits), extended-precision floating-point numbers (96 bits), or packed binary coded decimal (96 bits) depending on the qualifier.

Each memory unit allocated is set to the value specified in the directive. This directive causes the location counter to be aligned to a word boundary, unless the .B qualifier is specified.

### Example

DCB.L 100,\$FFFFFF

## DS — Defines Storage

### Syntax

[*label*] DS[.qualifier] *size*

### Description

*label*      Specifies a label that will be assigned the address of the first byte allocated.

*qualifier*      Defines the units in which storage is allocated. The units can be any of the following:

.B	Bytes
.W	Words
.L	Longwords
.S	Single precision
.D	Double precision
.X	Extended floating
.P	Packed binary coded decimal

The default is .W.

*size*      Specifies the number of units to be allocated by this directive. Any symbols used in this expression must be previously defined. The final expression cannot contain any relocatable terms.

The DS directive reserves a block of sequential locations of memory. It causes the program counter to be advanced and, for this reason, the contents of the reserved bytes are unpredictable. Locations can be reserved in units of bytes, words, longwords, single-precision floating-point numbers (32 bits), double-precision floating-point numbers (64 bits), extended-precision floating-point numbers (96 bits), or packed binary coded decimal (96 bits).

The DS directive causes the location counter to be aligned to a word boundary unless the .B qualifier is used. The form DS 0 can be used to force alignment between two DC.B, DS.B, or DCB.B statements, if necessary.

### Example

```
JAKE DS    $62
MOE   DS.B 100
```

## ELSEC — Specifies Conditional Assembly Converse

### Syntax

```
ELSEC
```

### Description

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 block of conditional assembly statements.

### Example

```
IFNE    MAIN
      .
      .
ELSEC
      .
      .
ENDC
```

## END — Ends Assembly

### Syntax

END [*expression*]

### Description

*expression*

Specifies an address placed in the end record of the load module that informs the linker 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 linker contains a starting address, execution begins at absolute 0.

The END directive informs the assembler that the last source statement has been read and indicates a load module starting address. Any statements following the END directive will not be processed.

If *expression* is not present but a comment field is present, the latter must be preceded by a semicolon (;) or an exclamation mark (!).

Specifying a load address in the END directive also informs the linker that this is a main program. If multiple load modules are combined by the linker, only one module can specify a load address and, for this reason, a main program is indicated.

**Example**

## Line Address

```

1           MAIN1    IDNT
2           *
3           *
4           XDEF    INBUF,ECHO
5           XREF    READ,SCAN
6           *
7           SECT.S  SECT1
8   00000000  INBUF   DS.B    80      ;Input buffer
9   00000050  ECHO    DS.B    1       ;Echo flag
10          *
11          ORG     $1000
12  00001000 6000 0002  BRA     PROC
13  00001004 2E7C 0000 1000  PROC    MOVE.L #$_1000,SP ;Set stack
14          *
15          SECT    SECT2
16  00000000 4EB9 0000 0000  E MAIN2  JSR     READ      ;Read next line
17  00000006 227C 0000 0000  R        MOVE.L #INBUF,A1 ;Start of buffer
18  0000000C 1219                 MAIN10  MOVE.B  (A1)+,D1
19  0000000E 0C01 0014             CMP.B   #BLNK,D1 ;Check for non-blank
20  00000012 67F8                 BEQ     MAIN10
21  00000014 4EB9 0000 0000  E        JSR     SCAN      ;Get value
22  0000001A 4E75                 RTS
23          *
24  00000014             BLNK    EQU     20
25          END    MAIN2

```

## Symbol Table

Label	Value
BLNK	00000014
ECHO	SECT1:00000050
INBUF	SECT1:00000000
MAIN10	SECT2:0000000C
MAIN2	SECT2:00000000
PROC	00001004
READ	External
SCAN	External

## ENDC — Ends Conditional Assembly Code

### Syntax

```
ENDC
```

### Description

The **ENDC** directive informs the assembler where the source code, which is subject to the conditional assembly statement, ends. In the case of nested **IFxx** statements, an **ENDC** is paired with the most recent **IFxx** statement.

### Example

```
MOVE  #22,D2
IFEQ  SUM-4
ORI   #200,D3      ; assembled if
ADD   D0,VALUE+3  ; SUM-4 is non-zero
ELSEC
ORI   #$1F,D3      ; assembled if
ROL   #1,D0        ; SUM-4 is zero
ENDC
```

In the example, 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 nonzero, the opposite occurs. To inhibit listing the nonassembled instructions, you can use the **OPT -I** directive.

## **ENDR — Ends Repeat**

### **Syntax**

```
ENDR
```

### **Description**

The ENDR directive ends a repeat statement as defined in the REPT or IRP or IRPC directives. Note that ENDR does not terminate a macro definition.

### **Example**

```
IRP D1
ADD D0, VALUE+3
ENDR
```

## EQU — Equates a Symbol to an Expression

### Syntax

*label EQU {expression | keyword}*

### Description

<i>label</i>	Defines symbol name.
<i>expression</i>	Sets the symbol to the value of <i>expression</i> for the duration of the current assembly. Any attempt to reequate the same label will result in an error.
	Any symbol used in <i>expression</i> must have been previously defined, externally defined, or defined as a simple forward reference. If <i>expression</i> uses previously defined symbols, the standard expression rules apply (see the <i>Expressions</i> section in Chapter 2, <i>Assembly Language</i> ). If <i>expression</i> uses an externally defined symbol, a constant number can be added or subtracted from the symbol. If the expression is defined as a forward reference, only one symbol can be in the expression and no operators are allowed.
<i>keyword</i>	Sets the symbol to the value of <i>keyword</i> , which is a keyword defined by the assembler or a symbol previously defined by the EQU directive as a keyword.

The EQU directive causes the assembler to assign a particular value to a new label. The new label can be an absolute symbol, a relocatable symbol, or even an external symbol. If a symbol is equated to an expression containing an external symbol, the expression cannot contain any other relocatable terms.

EQU also defines new keywords instead of the predefined assembler keywords. This lets you assign meaningful names to processor registers.

### Examples

```
SEVEN    EQU      D7      ; data register
INDEX    EQU      A5      ; address register
BLNK     EQU      20      ; ASCII value for a space
BUFPTR   EQU      BUFF    ; label at beginning of a buffer

                  XREF    xrefsymbol ; externally defined symbol
sym1     EQU      xrefsymbol+7
sym2     EQU      xrefsymbol-3
```

## FAIL — Generates Programmed Error

### Syntax

FAIL [*expression*]

### Description

*expression*      Defines the number of the FAIL directive. *expression* is a 32-bit absolute value and does not contain any forward references. The default value is 0.

The FAIL directive indicates an error or warning in the listing and error summary. 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 you intended.

When a FAIL directive is assembled, the assembler marks it with a fail encountered error message or warning and displays the value of *expression* in the address field of the listing. If the value of *expression* is 500 or more, a warning is generated; otherwise, an error message is generated.

**Example**

Command line: asm68k -l fail

Line	Address	
1	00000001	true EQU 1
2	00000000	false EQU 0
3		
4	00000000	test SET false ; this time pass
5		IFEQ test
6	00000000 4E71	NOP
7		ELSEC
8		FAIL 444
9		ENDC
10		
11	00000001	test SET true ; this time fail
12		IFEQ test
13		NOP
14		ELSEC
15	000001BC	FAIL 444 ; fail with error
** ERROR:(591) FAIL directive assembled.		
16		ENDC
17		
18	00000001	test SET true ; this time fail
19		IFEQ test
20		NOP
21		ELSEC
22	00000220	FAIL 544 ; fail with warning
** WARNING:(591) FAIL directive assembled.		
23		ENDC
24		END

## Symbol Table

Label	Value
false	00000000
test	00000001
true	00000001

Errors: 1, Warnings: 1

## FEQU — Equates a Symbol to a Floating-Point Expression

### Syntax

*label* FEQU[*.qualifier*] *floating\_point\_expression*

### Description

*label*      Defines a symbol name.

*qualifier*      Specifies the symbol type. *qualifier* can be:

.S      Single precision

.D      Double precision

.X      Extended precision

.P      Packed decimal

The default is .X.

*floating\_point\_expression*

Sets value of *label* for the duration of the current assembly. An attempt to reequate the same label will result in an error. Any symbols used in the expression must have been previously defined.

The FEQU directive assigns a floating-point expression to a symbol. ASM68K supports the IEEE standard floating-point number format with an optional exponent. Floating-point numbers can 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.

Underscores can 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 hexadecimal digits (up to 8 digits for single-precision or 16 digits for double-precision). The digits are placed in the field as they stand. You are 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.

### Examples

```
COUNT1  FEQU  3.14159
COUNT2  FEQU  -22E-100; ; Equivalent to -22 * 10-100
COUNT3  FEQU  123.45
COUNT4  FEQU  :9AB
```

## FOPT — Sets Floating-Point Options

### Syntax

```
FOPT ID=n
```

### Description

*n*

Specifies the coprocessor ID field. The range of *n* is 0 through 7.

The FOPT directive specifies the coprocessor ID field (0 through 7) used in subsequent 68881/882 floating-point instructions. If unspecified, the default 68881/882 coprocessor ID is 1.

### Examples

```
FOPT    ID=2      ; Specifies 68881/882 id # 2
FMOVE.D #2.0,FP0 ; Move to 68881/882 id # 2
FOPT    ID=1      ; Specifies 68881/882 id # 1
FMOVE.D #2.0,FP0 ; Move to 68881/882 id # 1
```

## FORMAT — Formats Listing

### Syntax

[NO]FORMAT

### Description

ASM68K does not require this directive but recognizes it for compatibility with the Motorola **FORMAT** directive.

Motorola uses the **FORMAT** directive to format the source listing. **NOFORMAT** prevents the formatting of the source listing.

## IDNT — Specifies Module Name

### Syntax

*name* IDNT

### Description

*name*

Specifies the module name which is passed to the linker. This name must follow all the rules of a symbol, and must appear in the label field of the statement. The operand field of the statement is ignored.

The IDNT directive assigns 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 you do not use the IDNT directive, the default object module name will be the same root name as the assembler input source file name (without path and extension).

### Example

LAB1 IDNT

## IFC — Checks if Strings Equal (Conditional Assembly)

### Syntax

```
IFC  [string1], [string2]
IFNC [string1], [string2]
```

### Description

- |                |  |
|----------------|--|
| <i>string1</i> | Represents a string of characters (see rules below). |
| <i>string2</i> | Represents a string of characters (see rules below). |

The IFC directive tests whether two strings are equal. Depending on the result of the comparison, statements up to the next ELSEC or ENDC will or will not be assembled (like the IF statement). This directive takes two string arguments, both optional, separated by a required comma.

The following rules are applied to *string1* (note that 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. 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 can be used 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 can 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.

The following rules are applied to *string2* (note that the term nonblank excludes tab characters also):

- 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 extends 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 extends 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 cannot contain semicolons.

A string delimited by quotes or up arrows is always unequal to a string not delimited by quotes, so it is not advisable to mix these two forms.

### Examples

```
IFC  'STRING','STRING'      ; Equal- 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)
IFC  string , string     ; Unequal (no terminating semicolon)
```

## IFDEF — Checks if Symbol Defined (Conditional Assembly)

### Syntax

```
IFDEF symbol
```

### Description

<i>symbol</i>	Represents any legal assembler symbol.
---------------	--

The IFDEF directive determines whether the symbol is defined or has been declared external. If the symbol has been defined or declared external, the IFDEF directive assembles code up to the next ENDC or ELSEC directive. No forward reference is allowed.

### Example

```
IFDEF LABEL
```

## IFEQ — Checks if Value Equal to Zero (Conditional Assembly)

### Syntax

`IFEQ expression`

### Description

*expression*

Evaluates to a value that determines whether or not the assembly between the **IFEQ** and the following **ELSEC** or **ENDC** will take place. Any symbols used in this expression must have been previously defined. The expression cannot be relocatable.

The **IFEQ** directive conditionally assembles source text between the **IFEQ** directive and the **ELSEC** or **ENDC** directive. When *expression* is equal to zero, the code will be assembled. For example, **IFEQ 5** will not cause the following code to be assembled while **IFEQ 0** will.

**IFEQ** statements can be nested up to 16 levels and appear at any place within the source text.

### Notes

The **IFEQ** directive performs a signed comparison, treating *expression* as a two's complement 32-bit signed integer ranging from -\$80000000 to +\$7FFFFFFF. In contrast, the logical operator = performs unsigned comparisons, treating operands as 32-bit unsigned integers ranging from 0 to +\$FFFFFF. Therefore, **IFEQ x** is not equivalent to **IFNE x=0**. Logical operators return a value of \$FFFFFF for true and zero for false. For information on expressions, refer to the section *Expressions* in Chapter 2, *Assembly Language*.

**Example**

```
Line Address
1                               opt not
2      00000001      ONE   EQU 1
3      00000000      ZERO  EQU 0
4                               IFEQ ZERO
5      00000000 4E71      NOP    ; assembled
6      00000002 4E71      NOP    ; assembled
7
8                               NOP    ; unassembled
9
10
11
12      IFEQ ONE      NOP    ; unassembled
13      NOP            NOP    ; unassembled
14
15      00000004 4E71      NOP    ; assembled
16      00000006 4E71      NOP    ; assembled
17
18      END
```

## IFGE — Checks if Value Non-Negative (Conditional Assembly)

### Syntax

IFGE *expression*

### Description

<i>expression</i>	Evaluates to a value that determines whether or not the assembly between the IFGE and the following ELSEC or ENDC will take place. Any symbols used in this expression must have been previously defined. The expression cannot be relocatable.
-------------------	---

The IFGE directive conditionally assembles source text between the IFGE directive and the ELSEC or ENDC directive. When *expression* is greater than or equal to zero, the code will be assembled. For example, IFGE 5 will cause the following code to be assembled while IFGE -5 will not.

IFGE statements can be nested up to 16 levels and appear at any place within the source text.

### Notes

The IFGE directive performs a signed comparison, treating *expression* as a two's complement 32-bit signed integer ranging from -\$80000000 to +\$7FFFFFFF. In contrast, the logical operator  $\geq$  performs unsigned comparisons, treating operands as 32-bit unsigned integers ranging from 0 to +\$FFFFFF. Therefore, IFGE *x* is not equivalent to IFNE *x* $\geq$ 0. Logical operators return a value of \$FFFFFF for true and zero for false. For information on expressions, refer to the section *Expressions* in Chapter 2, *Assembly Language*.

## IFGT — Checks if Value Greater Than Zero (Conditional Assembly)

### Syntax

`IFGT expression`

### Description

*expression*

Evaluates to a value that determines whether or not the assembly between the **IFGT** and the following **ELSEC** or **ENDC** will take place. Any symbols used in this expression must have been previously defined. The expression cannot be relocatable.

The **IFGT** directive conditionally assembles source text between the **IFGT** directive and the **ELSEC** or **ENDC** directive. When *expression* is greater than zero, the code will be assembled. For example, **IFGT 5** will cause the following code to be assembled while **IFGT -5** will not.

**IFGT** statements can be nested up to 16 levels and appear at any place within the source text.

### Notes

The **IFGT** directive performs a signed comparison, treating *expression* as a two's complement 32-bit signed integer ranging from -\$80000000 to +\$7FFFFFFF. In contrast, the logical operator **>** performs unsigned comparisons, treating operands as 32-bit unsigned integers ranging from 0 to +\$FFFFFF. Therefore, **IFGT x** is not equivalent to **IFNE x>0**. Logical operators return a value of \$FFFFFF for true and zero for false. For information on expressions, refer to the section *Expressions* in Chapter 2, *Assembly Language*.

## IFLE — Checks if Value Non-Positive (Conditional Assembly)

### Syntax

`IFLE expression`

### Description

*expression*

Evaluates to a value that determines whether or not the assembly between the `IFLE` and the following `ELSEC` or `ENDC` will take place. Any symbols used in this expression must have been previously defined. The expression cannot be relocatable.

The `IFLE` directive conditionally assembles source text between the `IFLE` directive and the `ELSEC` or `ENDC` directive. When *expression* is greater than or equal to zero, the code will be assembled. For example, `IFLE -5` will cause the following code to be assembled while `IFLE 5` will not.

`IFLE` statements can be nested up to 16 levels and appear at any place within the source text.

### Notes

The `IFLE` directive performs a signed comparison, treating *expression* as a two's complement 32-bit signed integer ranging from `-$80000000` to `+$7FFFFFFF`. In contrast, the logical operator `<=` performs unsigned comparisons, treating operands as 32-bit unsigned integers ranging from 0 to `+$FFFFFFF`. Therefore, `IFLE x` is not equivalent to `IFNE x<=0`. Logical operators return a value of `$FFFFFFF` for true and zero for false. For information on expressions, refer to the section *Expressions* in Chapter 2, *Assembly Language*.

## IFLT — Checks if Value Less Than Zero (Conditional Assembly)

### Syntax

IFLT *expression*

### Description

*expression*

Evaluates to a value that determines whether or not the assembly between the IFLT and the following ELSEC or ENDC will take place. Any symbols used in this expression must have been previously defined. The expression cannot be relocatable.

The IFLT directive conditionally assembles source text between the IFLT directive and the ELSEC or ENDC directive. When *expression* is greater than or equal to zero, the code will be assembled. For example, IFLT -5 will cause the following code to be assembled while IFLT 5 will not.

IFLT statements can be nested up to 16 levels and appear at any place within the source text.

### Notes

The IFLT directive performs a signed comparison, treating *expression* as a two's complement 32-bit signed integer ranging from -\$80000000 to +\$7FFFFFFF. In contrast, the logical operator <= performs unsigned comparisons, treating operands as 32-bit unsigned integers ranging from 0 to +\$FFFFFF. Therefore, IFLT *x* is not equivalent to IFNE *x*<=0. Logical operators return a value of \$FFFFFF for true and zero for false. For information on expressions, refer to the section *Expressions* in Chapter 2, *Assembly Language*.

## IFNC — Checks if Strings Not Equal (Conditional Assembly)

### Syntax

```
IFNC [string1], [string2]
```

### Description

*string1*                         Represents a string of characters (see rules below).

*string2*                         Represents a string of characters (see rules below).

The IFNC directive tests whether two strings are unequal. Depending on the result of the comparison, statements up to the next ELSEC or END C will or will not be assembled (like the IF statement). This directive takes two string arguments, both optional, separated by a required comma.

The following rules are applied to *string1* (note that 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. 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 can be used 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 can 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.

The following rules are applied to *string2* (note that the term nonblank excludes tab characters also):

- 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 extends 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 extends 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 cannot contain semicolons.

A string delimited by quotes or up arrows is always unequal to a string not delimited by quotes, so it is not advisable to mix these two forms.

### Examples

```
IFNC  'string',' string'      ; Unequal (blank in 2nd
      ; string) assembly continues
IFNC  'abc','abc'           ; Equal-assembley does not continue
```

## IFNDEF — Checks if Symbol Not Defined (Conditional Assembly)

### Syntax

```
IFNDEF symbol
```

### Description

*symbol*                      Represents any legal assembler symbol.

The IFNDEF directive determines whether the symbol is defined or has been declared external. If the symbol has not been defined or declared external, the IFNDEF directive assembles code up to the next ENDC or ELSEC directive. No forward reference is allowed.

### Examples

```
IFNDEF LABEL
```

## IFNE — Checks if Value Unequal to Zero (Conditional Assembly)

### Syntax

IFNE *expression*

### Description

*expression*

Evaluates to a value that determines whether or not the assembly between the IFNE and the following ELSEC or END<sup>C</sup> will take place. Any symbols used in this expression must have been previously defined. The expression cannot be relocatable.

The IFNE directive conditionally assembles source text between the IFNE directive and the ELSEC or END<sup>C</sup> directive. When *expression* is greater than or equal to zero, the code will be assembled. For example, IFNE 5 will cause the following code to be assembled while IFNE 0 will not.

IFNE statements can be nested up to 16 levels and appear at any place within the source text.

### Notes

The IFNE directive performs a signed comparison, treating *expression* as a two's complement 32-bit signed integer ranging from -\$80000000 to +\$7FFFFFFF. In contrast, the logical operator <> performs unsigned comparisons, treating operands as 32-bit unsigned integers ranging from 0 to +\$FFFFFFFF. Therefore, IFNE x is not equivalent to IFNE x<>0. Logical operators return a value of \$FFFFFF for true and zero for false. For information on expressions, refer to the section *Expressions* in Chapter 2, *Assembly Language*.

### **INCLUDE — Includes Source File**

## Syntax

**INCLUDE** *filename*

## Description

*filename* Names the assembler source file to be inserted.

The **INCLUDE** directive inserts an external source file into the input source code stream at assembly time. **INCLUDE** statements can be nested and can contain macro calls. A macro call can also contain an **INCLUDE** directive.

The **INCLUDE** file name is passed to the host operating system as specified without any lower- to upper-case conversion.

Default search paths can be set from the command line. The assembler will first search in the current directory for the named **INCLUDE** file and then in the specified -*i* (UNIX/DOS) or /**INCLUDE** (VMS) directories. For information on the correct syntax, see the *Microtec Research ASM68K User's Guide*.

## Example

INCLUDE EXTERNAL.SRC

## IRP — Specifies Indefinite Repeat

### Syntax

```
[label] IRP model_parameter[,actual_parameter]...
```

### Description

<i>label</i>	Assigns the address of the current program counter to <i>label</i> .
<i>model_parameter</i>	Specifies a parameter name.
<i>actual_parameter</i>	Specifies parameters that are to be substituted into the <i>model_parameter</i> .

The IRP directive repeats a sequence of statements enclosed by the IRP and ENDR directives once for each *actual\_parameter*. Each *actual\_parameter* is substituted in place of *model\_parameter*. Parameter substitution is identical to that which is performed in a macro. If no *actual\_parameter* is specified, the macro is expanded once with a null replacing the *model\_parameter*.

Like macro definitions, repeat directives cannot be nested. Only one macro definition can be used inside a repeat directive.

### Example

The following example shows an IRP directive, a model parameter, and actual parameters.

```
OPT M
XREF SUB1, SUB2, SUB3

; Three JSR instructions
; are generated
IRP DUMMY, SUB1, SUB2, SUB3
JSR DUMMY

ENDR
END
```

The resulting expansion is shown below:

Line	Address	
1		OPT M
2		XREF SUB1, SUB2, SUB3
3		
4		;
5		Three JSR instructions
6		;
7		are generated.
8		IRP DUMMY, SUB1, SUB2, SUB3
9		JSR DUMMY
9.1	00000000 4EB9 0000 0000 E	ENDR
9.2		JSR SUB1
9.3	00000006 4EB9 0000 0000 E	JSR SUB2
9.4		
9.5	0000000C 4EB9 0000 0000 E	JSR SUB3
9.6		
10		END

## IRPC — Specifies Indefinite Repeat Character

### Syntax

[*label*] IRPC *model\_parameter*[,*actual\_parameter*]

### Description

<i>label</i>	Assigns the address of the current program counter to <i>label</i> .
<i>model_parameter</i>	Specifies a parameter name.
<i>actual_parameter</i>	Specifies parameters that are to be substituted into the <i>model_parameter</i> .

The IRPC directive repeats a sequence of statements once for each character of *actual\_parameter*. The IRPC directive can be terminated with the ENDR directive. If no *actual\_parameter* is specified, the macro is expanded once with a null replacing the *model\_parameter*.

Like macro definitions, repeat directives cannot be nested. Only one macro definition can be used inside a repeat directive.

**Example**

```
Command line: asm68k -l irpcout
Line      Address
1          XREF  SUB
2
3          IRPC DUMMY,1234
4          MOVE #DUMMY,D0
5          ;Four MOVE and JSR instructions
generated
6          JSR  SUB
7          ENDR
7.1        00000000 303C 0001      MOVE #1,D0
7.2        ;Four MOVE and JSR instructions
generated
7.3        00000004 4EB9 0000 0000  E      JSR  SUB
7.4        0000000A 303C 0002      MOVE #2,D0
7.5        ;Four MOVE and JSR instructions
generated
7.6        0000000E 4EB9 0000 0000  E      JSR  SUB
7.7        00000014 303C 0003      MOVE #3,D0
7.8        ;Four MOVE and JSR instructions
generated
7.9        00000018 4EB9 0000 0000  E      JSR  SUB
7.10       0000001E 303C 0004      MOVE #4,D0
7.11       ;Four MOVE and JSR instructions
generated
7.12       00000022 4EB9 0000 0000  E      JSR  SUB
8          END
          Symbol Table
Label      Value
SUB       External
```

## LIST — Generates Assembly Listing

### Syntax

[NO]LIST

### Description

The LIST directive prints an assembly listing. The assembly listing is printed by default (LIST). When NOLIST is used, an assembly listing is not printed.

The OPT S directive is synonymous with the LIST directive.

## LLEN — Sets Length of Line in Assembler Listing

### Syntax

```
LLEN n
```

### Description

*n*

Specifies the number of characters in a line for the assembler listing; *n* must be between 37 and 1120 inclusive. The default line length is 132.

The LLEN directive changes the length of the line on the source listing. The value of 113 allows printing of the full 80 columns of the input source.

This directive does not affect the header lines at the top of each page, which are printed in a fixed-length format.

### Example

```
LLEN 128
```

## MASK2 — Generates Code to Run on MASK2 (R9M) Chip

### Syntax

MASK2

### Description

The MASK2 directive is recognized for Motorola compatibility. Its use will not generate an error, but the instruction is ignored.

## NAME — Specifies Module Name

### Syntax

NAME *modulename*

### Description

*modulename*      Specifies the module name within the object file.

The NAME directive assigns an internal module name to the object module produced by the assembler; the object file name is unchanged. 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 you do not specify a NAME or IDNT directive, the default module name is the input file name without path and extension.

### Notes

The module name will consist of every character up to the end of the line including blank spaces and comments.

## NOOBJ — Does Not Create an Output Object Module

### Syntax

NOOBJ

### Description

The NOOBJ directive suppresses creation of the output object module.

The OPT -O directive is synonymous with NOOBJ.

## OFFSET — Defines Table of Offsets

### Syntax

OFFSET *expression*

### Description

*expression*      Specifies new value for the location counter. This absolute expression cannot contain any references that make the expression relocatable.

The **OFFSET** directive defines a table of absolute offsets. It is present for convenience and Motorola 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 cannot contain instructions or any code-producing directives. **SET**, **EQU**, **REG**, **XDEF**, and **XREF** directives are allowed. **DC** and **DCB** directives are illegal within an **OFFSET** section. The **OFFSET** section must be terminated by an **ORG**, **OFFSET**, **SECT**, **SECTION**, **COMMON**, or **END** directive.

The typical use for **OFFSET** is to define a storage template in mnemonic terms. For example, suppose you want to define an array of 80 by 24 bytes representing a terminal screen. Since two dimensional arrays are not available in assembly language, the array must be defined as one dimensional: **SCREEN DS.B 80\*24**. You could set up the offsets **ROW1**, **ROW2**, etc., so that line I of the screen can be accessed as **ROWI**, as shown in the following example.

### Example

An example of template definition through the use of **EQU**:

```
SCREEN    DS.B     80*24
ROW1      EQU      SCREEN
ROW2      EQU      SCREEN+80
ROW3      EQU      SCREEN+160
.
.
.
END
```

The use of OFFSET for template definition provides a clearer alternative for complex structures:

```
    OFFSET 0
ROW1: OFFSET 80
ROW2: OFFSET 160
ROW3: OFFSET 240

    END
```

## OPT — Sets Options for Assembly

### Syntax

```
OPT [- | NO] [B | C | D | E | F | G | I | M | O | P | R | S | T | X  
| W | ABSPCADD | BRL | BRS | BRW | BRB | CASE | CEX | CL | CRE | FRL  
| FRS | MC | MD | MEX | NEST=n | OLD | OP=n | P=chip[/cotype] | PCO  
| PCR | PCS | QUICK | REL32]
```

### Description

ABSPCADD	An absolute expression appearing in conjunction with the mnemonic PC refers to an address rather than an absolute displacement. 5(PC) would refer to absolute address 5, using with PC-relative mode, rather than <i>5+current PC</i> . This option applies to the base displacement, not the outer displacement, in the 68020 expressions containing square brackets. This option can be turned on and off at your discretion. The last ABSPCADD setting applies. For more information on this option, see <i>Addressing Mode Syntax</i> in Chapter 3, <i>Instructions and Addressing Modes</i> . (default: ABSPCADD)
B	
BRS	
BRB	Forces forward references in relative branch instructions (Bcc, BRA, BSR) to use the short form of the instruction (8-bit displacement). (default: BRW)
BRL	Forces the long addressing mode to be used in relative branch instructions (Bcc, BRA, BSR) that have forward references. If NOBRL is specified, then the assembler is set to the default BRW. (default: BRW)
	For 32-bit processors, when OPT OLD has not been specified, 32-bit displacements are used. When OPT OLD has been specified, 16-bit displacements are used. In all other processor modes, 16-bit displacements are used. (default: BRW)
BRW	Forces 16-bit displacements to always be used in relative branch instructions (Bcc, BRA, BSR) that have forward references. BRW cannot be negated. (default: BRW)
C	

CEX	Lists all lines of object code that are generated by the DC directive. This option does not affect the DCB directive. CEX and c are synonymous. (default: CEX)
CASE	Retains case-sensitivity of symbols. For example, <b>LOOP5</b> is different from <b>loop5</b> . If NOCASE is used, all symbols will be converted to uppercase. (default: CASE)
CL	Lists instructions that are not assembled due to conditional assembly statements. CL and I are synonymous. (default: CL)
CRE	Lists the cross-reference table on the output listing. The CRE option overrides the symbol table output option, T. If both T and CRE are specified, a cross-reference table will be generated. CRE is a synonym for X. (default = NOCRE)
D	Places local symbols in the absolute or relocatable output object module, which is useful for debugging. This option must also be specified before any instruction that generates object code. If OPT CASE is used, symbols will be placed in the object module as defined. (default: NOD)
E	Lists lines with errors on your terminal as well as on the output listing. For a listing of just errors, you can turn off the source listing and only the errors will appear on the normal output device. (default = E)
F FRS	Forces instructions containing forward references to an absolute (nonrelocatable) address to use a 16-bit address (short form) instead of a 32-bit address. This option does not cause instructions to use an absolute addressing mode but applies to all instructions that can use absolute addressing modes, specifically excluding the Bxx instructions. FRS is a synonym for F, and FRL is a synonym for -F. (default: NOFRS)
FRL	Forces instructions containing forward references to an absolute (nonrelocatable) address to use a 32-bit address instead of a 16-bit address. FRL is a synonym for -F. (default: 32-bit address or FRL)
G	Lists assembler-generated symbols in the symbol or cross-

	reference table. If D is also set, these symbols are placed in the object module as well. (default: NOG)
I	Lists instructions that are not assembled due to conditional assembly statements. CL and I are synonymous. (default: CL)
M	
MEX	Lists macro and structured control directive expansions on the program listing. (default: MEX)
MC	Lists macro calls on the program listing. When the expansion of one macro contains a call to another macro, either NOM or NOMC will suppress listing the nested call. (default: MC)
MD	Lists macro definitions on the program listing. (default: MD)
NEST= <i>n</i>	Sets the nesting levels for macros. The default is set to the maximum number of nesting levels. (default: NEST=100 (UNIX/VMS) or NEST=20 (DOS))
O	Produces the output object module. (default: o)
OLD	Specifies that the interpretation of the BRL flag and the .L size qualifiers be 16-bit displacements for Bcc instructions when OPT BRL or OPT -B are specified, or when explicit .L qualifiers are used (as appropriate for the 68010 and earlier processors) even though the processor mode has been set to 68020. This is convenient for migrating 68000 programs onto 32-bit microprocessors (68020/30/40, 68EC020, 68EC030, 68EC040, and CPU32 family). (default: OLD)
OP= <i>n</i>	Resets the maximum number of optimization loops that the assembler will do if the -f opnop assembler command is set. The assembler will discontinue looping if there is a pass in which no optimization occurs. (default: OP=3)
P	
PCO	Uses the program counter with displacement addressing mode on backward references within the absolute (ORG) section, provided that this addressing 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. (default: NOPCO)

P=*chip* [ /*cotype* ]

Identifies the target processor and coprocessor. Valid *type* values include all 68000 family processors. Valid coprocessors (*cotype*) include: 68851 or 68881.

The 68881 coprocessor is compatible with all 68000 family processors with the exception of the 68040. The 68851 coprocessor is compatible with all 68000 family processors with the exception of 68030 and 68040.

### Example

P=68020

The P=*chip* option is distinguished from OPT P by the equal sign which must immediately follow the P. See the CHIP directive (which is equivalent to OPT P=) for a discussion of the differences between the various target processors.

The preceding NO or minus sign is not permitted with this P=*chip* option.

(default: 68000/68881)

PCR

Uses the program counter relative addressing mode on references from a relocatable section to the same section within the current module for all instructions for which this is a legal addressing mode. PCR differs from PCS in that PCS applies to all relocatable sections within this module and assumes you know the boundaries. PCR will affect only intrasection expressions within the current module (expressions for which the assembler has enough information to generate the correct relative offset). (default: PCR)

PCS

R

Uses the program counter relative addressing mode on backward references from a relocatable section to the same section or to a different relocatable section for all instructions for which this is a legal addressing mode. This does not affect forward references. If R is on and a backward reference within a relocatable section results in a displacement larger than 16 bits, it is considered an error. (default: NOPCS)

QUICK

Changes the MOVE, ADD, and SUB instructions to the more efficient MOVEQ, ADDQ, and SUBQ instructions.

Before the Change	After the Change
-------------------	------------------

MOVE.L #data, Dn	MOVEQ #data, Dn
ADD #data, ea	ADDQ #data, ea
SUB #data, ea	SUBQ #data, ea

where:

*data* Legal values are -128 to 127 for MOVE and MOVEQ instructions.

Legal values are 1 to 8 for ADD, ADDQ, SUB, and SUBQ instructions.

*ea* Effective address.

For more information on these instructions, see a *Motorola Microprocessor User's Manual*.  
(default: QUICK)

**REL32** Forces the assembler to default to 32-bit base and outer displacements when the address range is 32 bits long.

Addressing modes which first appeared with the 68020:

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

defaulted the size of the outer and base displacements to word for forward references, external references, complex expressions, and relocatable expressions. While the code produced was smaller, you had to always size cast displacement expressions if you had a processor that accessed a full 32-bit address range (68020/30/040 and CPU32 family). The REL32 flag lets you change the default to 32 bits without size casting displacement expressions. This flag can be turned on and off at any time in the program. It will only be effective if the processor type is set to 68020, 68030, 68040 or CPU32 family.

(default: NOREL32)

**s** Lists the source text on the output listing. The directives LIST and NOLIST are other ways to specify OPT s and OPT -s respectively.

(default: s)

**t** Lists the symbol table on the output listing.  
(default: t)

---

W	Prints warnings during the assembly. (default: w)
X	Lists the cross-reference table on the output listing. The x option overrides the symbol table output option, T. If both T and x are specified, a cross-reference table will be generated. CRE is a synonym for x. (default: NOCRE)

The OPT directive generates listings of the elements specified. It influences the assembler's choice of addressing modes in ambiguous situations and controls 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.
- The symbol table is not placed into the object module.
- References to unknown locations will use an absolute addressing mode unless you specifically request otherwise.
- Forward references and external references not associated with a section name will leave room for an absolute long address.
- A relative branch to a forward reference will use the long (32-bit displacement) form of the instruction.
- The target chip is the 68000.
- The 68881/882 instructions are legal.

To turn off an option, precede it by a minus sign (-) or the word NO.

Error messages are always listed, regardless of the elements specified. In particular, the E option can be used to list error messages on the standard output device.

### Example

```
OPT -CRE,D ; Does not list the cross reference table.  
             ; Puts the symbol table in the object module.
```

## ORG — Begins Absolute Section

### Syntax

```
ORG[.qualifier] [expression] [,name]
```

### Description

#### *qualifier*

Specifies the address form for instructions containing forward references to an absolute (nonrelocatable) address. The following values are legal:

- S ORG.S is interpreted as both ORG and OPT F.
- L ORG.L is interpreted as both ORG and OPT -F.

#### *expression*

Replaces the contents of the assembly program counter. Bytes subsequently assembled will be assigned memory addresses beginning with this value. This expression can contain no forward, undefined, or relocatable symbols (including external references).

The form \* {+ | -} *displacement* indicates an absolute value, where *displacement* is a constant number and \* indicates the ending value of the previous absolute section or 0 for the first absolute section.

#### *name*

Specifies the name of the section.

The ORG directive begins an absolute section. 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.

If the ORG directive is used and *expression* is not specified:

- The F option is unchanged (see the OPT directive in this chapter).
- The location counter is set to the address following the last preceding absolute section if there was one; otherwise, the location counter is set to 0.
- A semicolon (;) or exclamation mark (!) must precede comment fields.

All subsequent bytes of code will be assigned sequential addresses beginning with the address in the location counter.

### Example

```
ORG $100
```

## PAGE — Advances Listing Form to Next Page

### Syntax

[NO] PAGE

### Description

The PAGE directive instructs the assembler to skip to the top of the next page on the listing form. You may want to start each subroutine on a new page for readability. If the NOPAGE directive was previously specified, this directive is ignored.

The NOPAGE 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 reenabled.

## PLEN — Sets Length of Listing Page

### Syntax

PLEN *n*

### Description

*n*              Specifies the number of lines on an assembly listing page.  
This absolute expression must have a value greater than 12.

PLEN specifies the number of lines on an assembly listing page. The default value is 60.

### Example

PLEN 58

## REG — Defines Register List

### Syntax

*label REG register\_list*

### Description

*label*

Defines a symbol name.

*register\_list*

Specifies a list of registers in the format recognized by the **MOVEM** instruction. It can be any of the following:

- A single register.
- 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 occur first).
- Any combination of the above separated by a slash.

The **REG** directive assigns a symbolic name to a register list for future use by the **MOVEM** instruction. The name can be redefined during the assembly as a different register list.

### Example

```
SAVE REG A1-A5/D0/D2-D4/D7  
MOVEM (A6),SAVE
```

## REPT — Specifies Repeat

### Syntax

[*label*] REPT *count*

### Description

<i>label</i>	Assigns the address of the current program counter to <i>label</i> .
<i>count</i>	Specifies the number of times to repeat the code. This expression cannot be relocatable or contain symbols not previously defined.

The REPT directive repeats a sequence of directives 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 assembler encounters the REPT directive.

### Example

```
REPT 3      ; Repeat following lines (until ENDR encountered)
        ; 3 times
DC.B  'A'
DC.B  'B'
ENDR
```

## RESTORE — Restores Options

### Syntax

```
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=68020/68881 ; 68020 instructions are now legal
      .
      .
CAS D0,D1,(A3)    ; 68020 instruction
      .
      .
RESTORE
      .          ; 68020 instructions are no longer legal
      .
END
```

## SAVE — Saves Options

### Syntax

```
SAVE
```

### Description

The **SAVE** directive saves the current set of **OPT** options (see the **OPT** command for a list of these options).

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=68020/68881 ; 68020 instructions are now legal
      .
      .

CAS D0,D1,(A3)    ; 68020 instruction
      .
      .

RESTORE
      . . . ; 68020 instructions are no longer legal
      .
      .

END
```

## SECT / SECTION — Specifies Section

### Syntax

```
SECT[.S] {sname | snumber} [, [align][,type][,hptype]]
```

### Description

.S	Assigns short attribute to the section. All symbols specified will be found in an area of memory accessible by 16-bit addresses, or they will be constants with 16-bit or smaller values.
<i>sname</i>	Specifies the noncommon section name. Any valid section can be used.
<i>snumber</i>	Specifies the section number. Up to two decimal digits can be used.
<i>align</i>	Specifies the bytes of alignment, either 2 or 4. The section alignment attribute lets you specify that a section be located on a modulo 2 or modulo 4 boundary. For more information, see Chapter 4, <i>Relocation</i> .
<i>type</i>	Specifies the type of section. It can be:  C      Code D      Data M      Mixed code, data, etc. R      ROMable Data
<i>hptype</i>	Specifies the HP 64000 section type. It can be:  A      ABS P      PROG D      DATA C      COMN

**SECT** and **SECTION** are equivalent. The statements following the **SECT** directive will be assembled in the specified relocatable section. This 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.

You can alternate among the various sections with multiple section directives within one program. The assembler will maintain the current value of the location counter for each section.

Each section has a type, which does not affect its placement in memory. Possible types are **M** (mixed code, data, etc.), **C** (code), **D** (data), and **R** (ROMable data, constants). For more information, see *Section Type Attributes* in Chapter 4, *Relocation*.

The *hptype* refers to the HP 64000 section type. If it is not specified, ASM68K assigns a *hptype* field according to the following rule: code sections map to **PROG**, data sections map to **DATA**, and ROMable sections map to **COMN** subject to the following restrictions:

1. There can be at most one **PROG**, one **DATA**, and one **COMN** section for each module.
2. The second and subsequent sections assigned to **PROG**, **DATA**, or **COMN** are called extra sections. Extra sections have **ABS** in their *hptype* field, and their local symbols cannot be written to the HP *asmb\_sym* file.

You cannot override the first restriction by explicitly assigning more than one section to **PROG**, **DATA**, or **COMN**. If you do so, ASM68K will issue the following warning:

Maximum number of typed sections exceeded in HP mode

This warning is issued whenever an extra section is created regardless of whether the *hptype* was set explicitly or implicitly.

If *hptype* is **ABS**, the section must be treated as an extra section; that is, the section will have no symbols in the *asmb\_sym* file because it is impossible to relocate them. For more information on HP 64000 issues, refer to Appendix A, *HP 64000 Development System Support*, in your *ASM68K User's Guide*.

### Example

```
SECT      SECT1      ; Name is SECT1
SECT.S    CODE,,,P   ; Name is CODE, noncommon
           ; section, HP type is PROG
SECTION   0          ; Name is 0, noncommon section
SECT     A,4        ; First byte of section A is
           ; quad-aligned
SECT     B,4,C      ; quad-aligned, section type =
           ; program code
SECT     C,,D       ; C section type = data
```

## SET — Equates a Symbol to an Expression

### Syntax

*label SET expression*

### Description

<i>label</i>	Specifies symbol name.
<i>expression</i>	Assigns value to <i>label</i> until changed by another SET directive. Any symbols used in the expression must have been previously defined.

The SET directive sets a symbol equal to a particular value. Unlike the EQU directive, multiple SET directives for the same symbol can 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 can also be used to define new keywords.

### Example

```
GO SET 5  
GO SET GO+10
```

## SPC — Spaces Lines On Listing

### Syntax

SPC *expression*

### Description

*expression*      Specifies number of lines to skip on the output listing. The expression must evaluate to an absolute value. *expression* cannot be relocatable but it can contain forward references.

The SPC directive causes one or more blank lines to appear on the output listing. It lets you format the listing for readability. The directive itself does not appear in the listing.

You can also use a blank source statement to insert blank lines on the listing.

### Example

SPC 7

## TTL — Sets Program Heading

### Syntax

`TTL heading`

### Description

#### *heading*

The title to be placed at the beginning of each page. The heading can be up to 60 characters. The title is case-sensitive. Additional characters are ignored.

You can optionally delimit the heading with single quotes as shown in the example. The quotes are not considered part of the title. If the terminating quote is omitted, only the first 60 characters will be used.

The TTL directive prints a heading at the beginning of each page of the listing in addition to the standard header. The default heading defined by the assembler is blank. This directive must be the first statement in the program if you want a specified title to appear on the first page of the output listing.

### Example

```
TTL 'TEST PROGRAM'
```

## XCOM — Specifies Weak External Reference

### Syntax

```
XCOM symbol,size
```

### Description

*symbol*

Names a symbol referenced in this module but defined in a different module or by the linker.

*size*

Specifies the size in bytes that the linker will reserve if there is no specific public definition for this symbol.

The XCOM directive specifies a symbol that is referenced in this module but is assumed to be defined in a separate module and the symbol remains undefined at link time.

XCOM is a Microtec Research extension to the Motorola assembly language and is of limited use to the assembly language programmer. This directive was created to support the assembly of compiler-generated assembly code. Some languages, like C, permit the referencing of global data items declared outside the current module. In the case where all modules reference the item but none allocate space for it, the *size* qualifier lets the linker properly account for it.

XCOM is identical to XREF with the exception that only one symbol per line is allowed and a required size qualifier is added. Currently, section and short/long specification are not supported.

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 common reference is presumed to be absolute by default and is valid in certain constructs where only absolute values are permitted (e.g., *symbol(An)*).

### 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 — Specifies External Definition

### Syntax

```
XDEF symbol [, symbol] ...
```

### Description

*symbol*

Specifies an external symbol that can be referenced by other modules.

The XDEF 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 always placed in the relocatable object module. Spaces are not allowed in the symbol list.

XDEF directives can appear anywhere within the program. Symbols that are declared with this directive, but not defined in the program, will be flagged as undefined in the output listing. Symbols can be declared external multiple times.

### Example

```
XDEF SCAN, LABEL, COSINE
```

## XREF — Specifies External Reference

### Syntax

```
XREF [.S] [sectname:] symbol [, [sectname:] symbol] ...
```

### Description

.S

Assigns short attribute to the section. All symbols specified will be found in an area of memory accessible by 16-bit addresses, or they will be constants with 16-bit or smaller values.

sectname

Specifies section name or number in which the symbol is expected to be defined. The linker gives a warning if the symbol is defined in a section with a different name.

symbol

Names a symbol referenced in this module but defined in a different module.

The XREF directive specifies a list of symbols that are referenced in this module but defined in a separate module. External symbol references can appear in any section including absolute sections. XREF directives can appear anywhere within the program. You can declare symbols to be externally defined multiple times. Spaces are not allowed in the symbol list.

Specifying the section name (or number) of an external reference sometimes affects the assembler's choice of addressing mode (refer to Chapter 3, *Instructions and Addressing Modes*, for more information). Also, during the linking process, the linker 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 by default and is valid in certain constructs where only absolute values are permitted (e.g., *symbol(An)*).

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 XREF statement. It is legal for a section name that has not been defined to appear in XREF statements. In this case, however, the section name counts toward the maximum allowable total sections. For more information on program sections, refer to *Program Sections* in Chapter 4, *Relocation*, in this manual.

### Example

```
XREF sym1,sym2,sect1:sym3,2:sym4
```

## Overview

A macro is a sequence of instructions that may be automatically inserted in the assembly source text by encoding a single instruction — the macro call. A macro is defined once and can be called any number of times. It can contain parameters that can be changed for each call. The macro facility simplifies the coding of programs, reduces the chance of user error, and makes programs easier to understand.

Macro functions can be altered by changing the source code in only one location: the macro definition. A macro definition consists of three parts: a heading, a body, and a terminator. This definition must precede any macro call.

A macro can be redefined at any place in the program; the most recent definition is used when the macro is called. A standard mnemonic (e.g., OR) can also be redefined by defining a macro with the same name. In this case, all subsequent uses of that instruction in the program cause the macro to be expanded.

### Macro Heading

The heading, which consists of the directive **MACRO**, gives the macro a name and defines a set of formal parameters. For more information on the macro heading, see the **MACRO** directive in this chapter.

### Macro Body

The first line of code following the **MACRO** directive is the start of the macro body. The macro statements that make up the macro body are placed in a macro file for use when the macro is called. During a macro call, an error is generated if another macro is defined within the macro.

No statement in a macro definition is assembled at definition time. Statements in the macro definition are 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 can appear within the macro body in any field. When the macro is called, all parameter names appearing in the macro body will be substituted by the actual parameter values from the macro call. Parameters can exist anywhere in the macro body, even in a comment statement or in the comment field of a statement.

A formal parameter in the macro body is indicated by a backslash (\). When referring to macro parameters in the macro body, you can precede the macro parameter with &&. This lets you embed the parameter in a string.

## Macro Terminator

The ENDM directive terminates the macro definition. During a macro definition, an ENDM directive must be found before another MACRO directive can be used.

## Calling a Macro

### Syntax:

*[label] name[.qualifier] parameter[,parameter] ...*

### Description:

*label*                   Assigned the current program counter value to *label*.

*name*                   Specifies the name of the macro. This name should have been defined by the MACRO directive or an error message will be generated.

*qualifier*           An optional qualifier that is passed to the macro as parameter \0.

*parameter*           Specifies constants, symbols, expressions, character strings, or any other text separated by commas. The maximum number of parameters supported in a macro call is 35.

A macro can be called by encoding the macro name in the operation field of the statement. The parameters in the macro call are actual parameters, and their names may be different from the formal parameters used in the macro definition. The actual parameters are 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 either the default value assigned in the macro definition or null (i.e., contain no actual characters). The formal parameter corresponding to a null actual parameter is removed during macro expansion. Any parameter not specified will be null. The parameter list is terminated by a blank, a tab, or a semicolon.

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 a macro expansion, it will also have the new value

after the expansion. SET directives within a macro body may alter the value of parameters passed to the macro.

The angle brackets (< >) delimit actual parameters that can 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 can 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 can consist of the angle brackets with no intervening characters, but the characters < and > cannot be passed as parameters and the parameter \0 cannot contain angle brackets.

The operator double equal sign (==), pronounced exists, can 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 false value (all zeros) otherwise.

The == operator can 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 to the end of the line. Therefore, using other operators to the right of == is useless. If a comment field follows an == operator, it must be prefixed by a semicolon (;). A parameter consisting entirely of blank characters will test null.

### Example:

An example of a macro call and its expansion is shown below:

```
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
```

Note that expanded code is marked with plus signs for description purposes only in the documentation.

```
MSET    MACRO  DATA, MEM
        IFNE   ==MEM
        MOVE   #DATA, MEM
        ELSEC
        MOVE   #DATA, (A1)
        ENDM
```

The above example checks whether the second parameter `MEM` is present. The `==` operator determines if there is a value for `MEM`, or if it is null. The `IFNE` instruction checks for a nonzero value. Therefore, if `MEM` has a nonzero value, the following `MOVE` instruction is executed; if `MEM` has a zero value (i.e., the parameter is not present in the macro call), the `MOVE` instruction following the `ELSEC` instruction is executed.

## NARG Symbol

The special symbol `NARG` is used to represent the number of nonnull 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 definition. It is typically used when generating tables within macros, along with conditional assembly statements.

**Example:**

```

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:**

```

        IF      NARG          ;(Value of NARG)
7F02   DC.B    ADD1, NARG
        GEN    ADD2,
        IF      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 is displayed there. In the example above, the **DC.B** directive is used so that the value of **NARG** can be seen.

**Macro Directives**

The assembler macro directives in the following section are organized alphabetically. An alphabetical listing of these directives is shown in Table 6-1.

**Table 6-1. Alphabetical Listing of the Macro Directives**

Macro Directive	Function
ENDM	Terminates Macro Definition
LOCAL	Defines Local Symbols
MACRO	Enters Macro Definition
MEXIT	Exits Macro

## ENDM — Terminates Macro Definition

### Syntax

[*label*] ENDM

### Description

*label*

Specifies the symbolic address of the first byte of memory following the inserted macro. If *label* has embedded parameters, it must be placed on the preceding line.

The ENDM directive terminates the macro definition. During a macro definition, an ENDM must be found before another MACRO directive can be used. An END directive that is found during a macro definition will terminate the definition as well as the assembly.

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.

## LOCAL — Defines Local Symbol

### Syntax

```
LOCAL symbol[,symbol] . . .
```

### Description

*symbol*

Defines a symbol local to this macro.

All labels, including those within macros, are global (i.e., 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, you can 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 can therefore be used in the operand field of instructions. The operand field of the LOCAL directive cannot contain any formal parameters defined on the MACRO directive line. As many LOCAL directives as necessary can 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 any 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 127 characters, or the assembler cannot 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 can be mixed with named local symbols without conflict although this is not recommended.

Local symbols declared by the \@ construction cannot 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.

**Example**

```
WAIT      MACRO      TIME      ;macro definition
          LOCAL
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
```

In the example above, expanded code is marked with plus signs for description purposes only in the documentation.

## MACRO — Enters Macro Definition

### Syntax

```
label MACRO parameter[,parameter]...
...
[label] ENDM
```

### Description

<i>label</i>	Specifies the macro name. The name cannot contain a period. A period in the macro name (i.e. ABC.w) will cause the assembler to search for a size qualifier following the period and apply it to the macro definition. Other than this condition, the macro name can be any legal symbol and it can 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.
<i>parameter</i>	Specifies a symbol known only to the macro definition; it can be used as a regular symbol outside the macro. Multiple parameters must be separated by commas.

The first line of code following the **MACRO** directive that is not a **LOCAL** macro directive is the start of the macro body. No statements in a macro definition are assembled at definition time. They are simply stored internally until called, at which time they are inserted in the source code at the position of the macro call. During a macro call, an error will be generated if another macro is defined within 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 a standard instruction mnemonic. A macro name can also be redefined as a new macro with a new body.

Unnamed (i.e., null) formal parameters are not allowed if they are followed by any named parameters.

### Example

```
XYZ MACRO , ,PARAM3
```

The above example is illegal because the null formal parameter is followed by PARAM3 which is a named parameter. This means that unnamed parameters must either come after all named parameters on the macro definition line or must be assigned a dummy name. Dummy formal parameters can be specified in the order in which they will occur on the macro call.

The name of a formal parameter specified on the MACRO directive can 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 (;).

Unnamed parameters and named parameters can be referenced with the Motorola backslash notation (\n where n is a nonnegative integer or string) in terms of the parameter's position on the call line. Parameter \0 is the qualifier (extension) of the macro call and can 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, etc. 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 for ASCII or an E for EBCDIC, macro parameters are not expanded within the string. This extension permits backslashes and formal parameter names to appear as a string.

When referring to macro parameters in the macro body, you can precede the macro parameter with &&. This lets you embed the parameter in a string.

### Example

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

## MEXIT — Exits Macro

### Syntax

```
[label] MEXIT
```

### Description

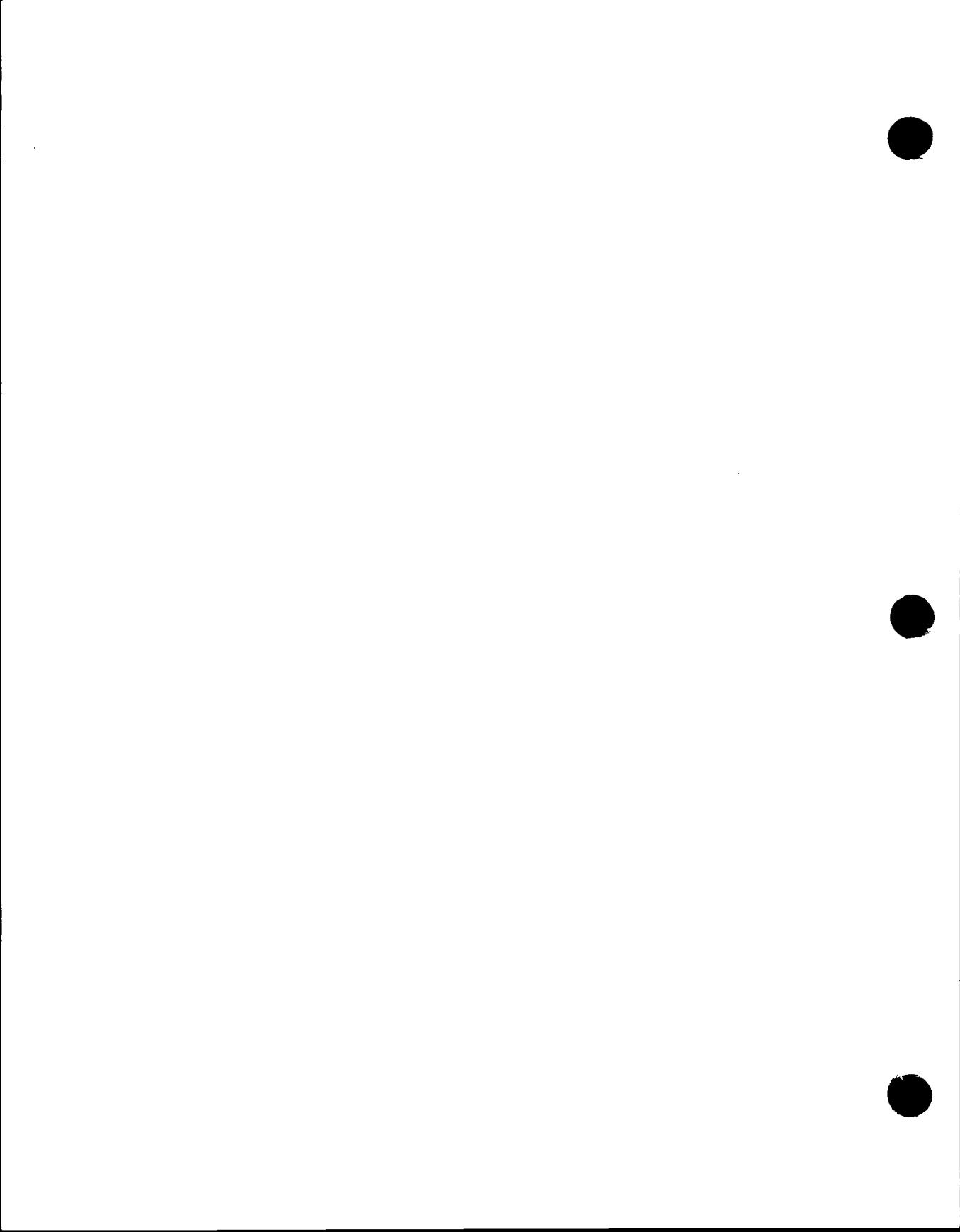
<i>label</i>	Assigns the address of the current location counter to <i>label</i> .
--------------	---

The **MEXIT** directive is 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** terminates a macro expansion, but only **ENDM** terminates a macro definition.

### Example

In the following example, the code following **MEXIT** will not be assembled if **DATA** is not zero.

```
STORE MACRO DATA
    .
    .
    .
    IF     DATA
    MEXIT
    ENDC
    .
    .
    .
ENDM
```



# Structured Control Directives 7

---

## Introduction

ASM68K includes several high-level language constructs similar to C constructs which control run-time loops and conditional execution. These constructs are provided to increase the ease with which you write code in assembly language while retaining most of the size and speed advantages inherent in the assembler language.

## Structured Control Expressions

Syntax for the **IF**, **UNTIL**, and **WHILE** statements require a field referred to as a structured control expression. 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.

### Example:

<MI>

This type of structured control expression generates a conditional branch instruction (**Bcc**), which tests the indicated bits of the condition codes. Any of the 14 condition codes accepted in the conditional branch instruction (**Bcc**) is legal and should be set previously. The test can be complemented to reflect your intent in some constructs. The expression is true if the condition code setting described is true.

2. Two expressions (as defined in Chapter 2, *Assembler Language*), separated by a condition code enclosed in angle brackets (e.g., **COUNT <LE> #4**).

This type of structured control expression generates a **CMP** (compare) instruction followed by a conditional branch. If the two expressions 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.

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.

3. Two structured control expressions, based on the two rules above, separated by the keywords **AND** or **OR**. These keywords can optionally have one of the qualifiers **.B**, **.W**, or **.L** (e.g., **COUNT <LE> #4 AND.B <CC>**).

This 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 **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 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).

Operands may be relocatable expressions. They cannot be complex. This is described in more detail under *Relocatable Expressions* in Chapter 4, *Relocation*. More complex combinations such as **COUNT <LE> #4 AND <CC> OR X <GT> Y** are not legal. At least one space or tab must appear between different parts of a structured control expression.

The following rules are applied:

1. The size of the **CMP** instruction (if any) for the expression to the left of the **AND** or **OR** is taken from the qualifier on the directive.
2. The size of the **CMP** instruction (if any) for the expression to the right of the **AND** or **OR** is taken from the qualifier on the **AND** or **OR**.
3. A compound expression containing **AND** is true if the expressions on both sides of **AND** are true, otherwise it is false.
4. 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 typically 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 codes as shown in Table 7-1.

**Table 7-1. Conditional Code Comparisons**

<b>Condition Code Conversion</b>	<b>New Condition</b>
<CC> to <LS>	Equivalent.
<CS> to <HI>	Equivalent.
<EQ> to <EQ>	Equivalent.
<NE> to <NE>	Equivalent.
<GE> to <LE>	Equivalent.
<GT> to <LT>	Equivalent.
<PL> to <MI>	Marked with a W (warning) flag on the assembly listing when they are not equivalent.
<VC> to <VC>	Marked with a W (warning) flag on the assembly
<VS> to <VS>	Marked with a W (warning) flag on the assembly listing when they are not equivalent

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 vice versa 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 your intent.

## Loop Controls

The assembler may generate local labels to handle branch instructions. These labels will start with two question marks (??). These labels will not show up in the symbol table unless OPT G is used.

### Example:

Line	Address	
1		
2		OPT G
3		
4		
4.1	00000000 123C 0001	FOR.B D1 = #1 TO #10 DO.S
4.2	00000004 6002	MOVE.B #1,D1 ;>> FOR <<
4.3	00000006 5201	BRA.S ??0001 ;>> FOR <<
4.4	00000008 0C01 000A	??0002 ADD.B #1,D1 ;>> FOR <<
4.5	0000000C 6E04	??0001 CMP.B #10,D1 ;>> FOR <<
5	0000000E 34C1	BGT.S ??0004 ;>> FOR <<
6		MOVE.W D1,(A2)+
6.1	00000010 60F4	ENDF
6.2		??0003 BRA ??0002 ;>> ENDF <<
7		??0004 ;>> ENDF <<
		END
		. . .
		. . .

### Symbol Table

Label	Value
??0001	0:00000008
??0002	0:00000006
??0003	0:00000010
??0004	0:00000012

Specific registers or memory locations are not predefined to hold the loop counts or values to be compared for the loop end conditions. At assembly time, you have control over which registers and memory locations are used. This is accomplished through operands specified for the constructs.

Unlike most high-level languages, there is no restriction on overwriting the loop counter, loop increment variable, or either of the loop bounds for the loop. When writing code for the loop body, special care should be taken not to alter these variables.

A SET variable cannot effectively be used as a loop index since SETs only affect variable values at assembly time.

## Structured Control Directives

Each of the structured control directives generates one or more assembly language instructions, which typically includes compare and branch instructions. These statements are used to control looping or conditional execution at runtime, not at assembly time. Table 7-2 lists the structured control directives.

**Table 7-2. Alphabetical Listing of the Structured Control Directives**

Directive	Function
BREAK <sup>1</sup>	Exits Loop Prematurely
FOR ... ENDF <sup>2</sup>	Specifies For Loop
IF ... ELSE ... ENDI <sup>2</sup>	Specifies If-Else Construct
NEXT <sup>1</sup>	Proceeds to Next Iteration of Loop
REPEAT ... UNTIL <sup>2</sup>	Specifies Repeat Loop
WHILE ... ENDW <sup>2</sup>	Specifies While Loop

1. Microtec Research has extended the Motorola control directives to add two directives that alter the flow of loop constructs.
2. The following keywords can be used within this construct: AND, BY, DO, DOWNTO, OR, THEN, and TO.

The IF structure directive should not be confused with the IF conditional assembly directive. At assembly time, each structure directive is translated into the appropriate assembly language code that will be executed at runtime. Conditional assembly directives do not generate code. They only control what will and will not be assembled.

## BREAK — Exits Loop Prematurely

### Syntax

**BREAK [ .*extent* ]**

### Description

*extent*

Specifies size (.S or .L) of the forward branch. 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 **BREAK** directive provides a convenient way to exit a **FOR**, **WHILE**, or **REPEAT** loop before the condition terminating the loop becomes true. **BREAK** generates a jump to the assembler-generated label (not known to you 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 can 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. Note that **BREAK** is not allowed in an **IF** construct.

## FOR . . . ENDF — Loops Based on Counter

### Syntax

```
FOR[.qualifier] op1 = op2 {TO | DOWNT0} op3 [BY op4] DO[.extent]
    loop_body
ENDF
```

### Description

<i>qualifier</i>	Applies this size qualifier to MOVE, CMP, ADD, or SUB instructions within the loop. Valid values for <i>qualifier</i> are: B, W, or L.
<i>op1</i>	Specifies the loop counter, which must be an expression that is legal as the right side of a MOVE instruction (typically a label or a register).
<i>op2</i>	Specifies initial value of the loop counter.
<i>op3</i>	Specifies final value of the loop counter.
<i>op4</i>	Increments (for TO) or decrements (for DOWNT0) loop counter by this value. If <i>op4</i> is not specified, the loop is incremented or decremented by 1.
<i>extent</i>	Specifies size of branch (.S or .L). If not present, the size of the forward branch is determined by the current setting of the B option (OPT BRL or OPT BRS).
<i>loop_body</i>	Specifies one or more statements. Any structured control statements must be properly nested.

This statement is an iterated loop, like the FOR loop of C and the DO loop of FORTRAN. The loop is executed until *op1* is greater than *op3* for TO (*op1* less than *op3* for DOWNT0), which means that it can be executed zero times if *op1* is greater than *op3* (for TO) when the loop is entered.

The FOR . . . ENDF loop generates a MOVE, a CMP, and either an ADD or SUB, plus various conditional and unconditional branches. 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.

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 high-level languages, there is no restriction on storing into the loop counter, loop increment, or either of the loop bounds within the loop (although doing so is error-prone).

Spaces or tabs are required as separators (including around an equal 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

```
FOR.B D1 = #1 TO #10 DO.S
  MOVE.W D1,(A2) +
ENDF
```

```
FOR.L A1 = #HIGHADD DOWNTO #LOWADD BY #4 DO
  MOVE.L (A1),-(A2)
ENDF
```

## IF . . . THEN . . . ELSE . . . ENDI — Performs Conditional Execution

### Syntax

```
IF[.qualifier] <structured_control_expression> THEN[.extent]
    then_part
    [ ELSE[.extent]
        else_part ]
ENDI
```

### Description

#### *qualifier*

Applies size qualifier to the *structured\_control\_expression*. Valid values for *qualifier* are: B, W, or L.

#### *structured\_control\_expression*

Specifies an expression that evaluates to a logical value of true or false. See the *Structured Control Expressions* section in this chapter for more information. Angle brackets around *structured\_control\_expression* are required.

#### *extent*

Specifies size (.S or .L) of the conditional branch. If not present, the size of the forward branch is determined by the current setting of the B option (**OPT BRL** or **OPT BRS**).

#### *then\_part*

Specifies a set of statements that will be executed if the *structured\_control\_expression* evaluates to true.

#### *else\_part*

Specifies a set of statements that will be executed if the *structured\_control\_expression* evaluates to false.

Only the statements in the *then\_part* will be executed if the *structured\_control\_expression* is true, and only the statements in the (optional) *else\_part* will be executed if the *structured\_control\_expression* is false.

The *qualifier* on the IF statement is used when generating code for the *structured\_control\_expression*. The *extent* code on THEN, which can 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*.

## Examples

```
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 IF's.
; As you see, the combination is difficult to understand
; sometimes.

IF 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
ELSE                 ; 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
```

## NEXT — Proceeds to Next Loop Iteration

### Syntax

`NEXT[ .extent]`

### Description

*extent*

Specifies size (.S or .L) of the forward branch. 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 NEXT directive provides a convenient way to proceed to the next iteration of a FOR, WHILE, or REPEAT loop. 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 can 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. Note that NEXT is not allowed in an IF construct.

## REPEAT . . . UNTIL — Loops Until Condition Not True

### Syntax

```
REPEAT
    loop_body
UNTIL[.qualifier] <structured_control_expression>
```

### Description

*qualifier*      Applies size qualifier to the *structured\_control\_expression*.

*loop\_body*      Specifies one or more statements that will be executed until the *structured\_control\_expression* evaluates to true. Any structured control statements must be properly nested.

*structured\_control\_expression*

Specifies an expression that evaluates to a logical value of true or false. See the *Structured Control Expressions* section in this chapter for more information. Angle brackets around *structured\_control\_expression* are required.

The test is placed at the end of the loop so that *loop\_body* is executed once even if *structured\_control\_expression* is true upon entry to the loop. The loop is executed until *structured\_control\_expression* becomes true.

The **REPEAT** generates only a label, and **UNTIL** generates code for *structured\_control\_expression*. Since all branches involved are backwards, there is no need for an extent field. A comment field on **UNTIL** must be delimited by a semicolon or exclamation point so the assembler will know to stop parsing *structured\_control\_expression*.

### Examples

```
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 — Loops While Condition True

### Syntax

```
WHILE[.qualifier] <structured_control_expression> DO[.extent]  
    loop_body  
ENDW
```

### Description

*qualifier*              Applies size qualifier to the *structured\_control\_expression*.

*structured\_control\_expression*

Specifies an expression that evaluates to a logical value of true or false. See the *Structured Control Expressions* section in this chapter for more information. Angle brackets around *structured\_control\_expression* are required.

*extent*              Specifies size (.S or .L) of the conditional branch. 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 *loop\_body* is repeated as long as *structured\_control\_expression* remains true. If it is false upon loop entry, then the loop body is not executed, but the CMP test is executed once and the condition codes will reflect this.

The *extent* field of the DO is applied to the conditional branch from the test out of the loop, a forward reference.

### Examples

```
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
```

## Structured Directive Nesting

Structured directives can be nested to create multi-level control structures subject to one 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 **O** (opcode) error and ignored by the assembler. If a terminating directive is omitted, an undefined label (**U**) error will follow the control directive beginning the high level construct.

An example of legal nesting is shown below.

### Example:

```
REPEAT
    MOVE.B (A1)+,NEXT_CHAR           ; fetch character
    IF.B #CR <EQ> NEXT_CHAR THEN.S   ; done if carriage return
        BREAK.S
    ELSE.S
        IF.B #BLANK <EQ> NEXT_CHAR THEN.S ; skip blanks
            NEXT.S
        ELSE.S
            MOVE.B NEXT_CHAR, (A2)+       ; copy character into buffer
            IF.L A2 <GT> #120 THEN.S
                JSR ERROR
                BREAK
            ENDI
        ENDI
    ENDI
    UNTIL A1 <GT> #120             ; Jump here on NEXT
                                    ; Jump here on BREAK
    RTS
```

## Structured Directive Listings

The code generated by structured control directives is shown in the same way on a listing as macro expansions. The code is marked with plus signs (+) and is not shown if the M or MEX option is turned off.

# Sample Assembler Session 8

## Introduction

As previously stated, the ASM68K 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 assembly program counter can 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.

At the end of the assembly process, a symbol table or cross-reference table can be generated.

## Assembler Listing

During pass two of the assembly process, a program listing is produced. The listing displays all information pertaining to the assembled program — both assembled data and your original source statements.

The main purpose of the listing is to convey all pertinent information about the assembled program, the memory addresses, and their values. It can be used as a documentation tool by including comments and remarks that describe the function of the particular program section. The link module, also produced during pass two, contains the object code address and value information in computer-readable format.

A sample listing is provided in Figure 8-1. Refer to the following points in order to examine and understand the listing.

1. The page headings on this sample show the time and date of the program run. This information might not be present in some installations.
2. The line titled `Command line:` specifies the command line used to invoke the assembler.

3. The source listing contains the columns titled Line and Address.

The Line column 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 Address column contains a value that represents the first memory address of any object code generated by this statement or the value of an EQU, SET, or FAIL directive.

4. 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.

5. To the right of the data words are the assembler relocation flags:

R      Relocatable operand  
E      External operand  
C      Complex and relocatable operand

If one operand is relocatable and another external, an E will be displayed.

6. Original source statements are reproduced to the right of the above information. Macro expansions and code generated by the structured-syntax directives are preceded with a plus (+) sign.
7. 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.

In the sample listing, there are no errors or warnings. However, if the assembler detects error conditions during the assembly process, a line titled ERROR will contain error code(s) describing the errors in the associated line of source code. Also, at the end of the listing, the assembler prints the message Errors: nn, Warnings: mm. Warnings are represented by a WARNING flag. Errors are represented by an ERROR flag.

The assembler substitutes two words of NOP's when it cannot translate a particular opcode and so provides room for patching the program. An explanation of the individual assembler errors is in Appendix B, *Assembler Error Messages*.

## Cross Reference Table Format

The cross-reference option is turned off by default. To turn it on, use **OPT CRE**. To turn it off again, use **OPT -CRE**. The assembler will produce either a cross-reference table or a symbol table, not both. The cross-reference table will be produced only if **OPT CRE** has been specified. Otherwise, a symbol table will be produced.

References can be accumulated for selected portions of the program by turning the cross-reference option on and off at the respective places in the program. However, to obtain the cross-reference listing, the option must be turned on before the **END** directive. Typically, the **OPT CRE** directive will be one of the first statements in the source program and will never be turned off.

An example of the cross-reference output is shown on the sample listing at the end of this chapter. Refer to the following points in order to examine and understand the cross-reference table format.

8. All user-defined symbols in the program are listed under the heading **Label**.
9. The symbol values are listed under **Value**. Any flag to the left of the values indicate the relocation type of the symbols.
10. 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 can appear for the symbol.

Section names, macro names, and the module name do not appear in the symbol table listing.

## Sample Program Listing

```
1 → Microtec Research ASM68K Version x.y   Wed Jul 22 14:08:55 1992  Page 1
2 → Command Line: /usr4/engr.sun4/bin/asm68k -l sample_asm.src
Line      Address
1
2
3
4
5
6
7
8
9
10
11
12
13
14      00000100
15
16      01000000 4E71
17
18
19
20
21      00000000 4E71
22
23
24      00000000 4E71
25
26
27
28
29
29.1
29.2
29.3
29.4
29.5
29.6

*****
*          *
*   Sample Standard 68000 Family Testcase   *
*          *
*****  
*   68000 ASSEMBLER SAMPLE TEST CASE
*       LLLEN    97
*       PLEN     48
*       OPT      CRE      ; Turn on cross reference
*
*DEFINE SOME SYMBOLS FOR LATER USE
*
*       ORG      $100
AVAL   DS      1      ;Absolute symbol
*       ORG      $fffffff
AVALHI NOP      ;Absolute symbol requiring
; 32 bits
; Note rounding of location
; counter to even address
RVAL1  SECT    SEC1
       NOP      ;Relocatable symbol in
; same section
       6 (
RVAL2  SECT    SEC2
       NOP      ;Relocatable symbol in
; different section
SECTION SEC1      ;Back to first relocatable
; section
XREF    EVAL      ;External reference
INCLUDE incl1.src *Test include directive
;
; This is a sample include file
;
*
*Allow errorless assembly of 68010 instructions
CHIP    68010
```

Figure 8-1. Sample Assembly Listing

Line		Address	
29.7			*
29.8			* C.1 bit manipulation, MOVEP, MOVES,
29.9			* IMMEDIATE INSTRUCTIONS
29.10			*
4 → 29.11	00000002	0030 00FF 0100	ORI.B #255,AVAL
29.12	00000008	003C 00FF	ORI.B #255,CCR
29.13	0000000C	0065 2710	ORI.W #10000,-(A5)
29.14	00000010	007C 00FF	ORI.W #255,SR
29.15	00000014	0098 0000 A7F8	ORI.L #43000,(A0)+
29.16	00000014	0104 000A	MOVEP 10(A2),D8
29.17	0000001E	0110	BTST D8,(A8)
29.18	00000020	0150	BCHG D8,(A8)
29.19	00000022	018A 000A	MOVEP D8,10(A2)
29.20	00000025	0190	BCLR D8,(A8)
29.21	00000028	01D0	BSET D8,(A8)
29.22	0000002A	0200 0041	ANDI.B #^A,D8
29.23	0000002E	023B 00FF 0100	ANDI.B #255,AVAL
29.24	00000034	023C 00FF	ANDI.B #255,CCR
29.25	00000038	0240 4E20	ANDI #20000,D8
29.26	0000003C	027C 00FF	ANDI #255,SR
29.27	00000040	0289 FFFF FFFF E	ANDI.L #\$FFFFFFFFFF,EVAL
5 → 29.28	0000004A	036B 000A	
29.29	0000004E	0407 00FF	MOVEP.W D1,10(AB)
29.30	00000052	0441 1000	SUBI.B #255,D7
29.31	00000055	045F 0000 9C40	SUBI #4096,D1
29.32	0000005C	0520 00FF	SUBI.L #40000,(A7)+
29.33	00000058	0540 0064	ADDI.B #255,-(A8)
6 → 29.34	00000064	0579 00FF 0000 E	ADDI.D #100,D0
		0000	ADDI.W #255,EVAL
30			END

**Figure 8-1. Sample Assembly Listing (cont.)**

7, 8,  
9, 10 →

Microtec Research ASM68K Version x.y Wed Jul 22 14:08:55 1992 Page 3

Cross Reference			
Label	Value	References	
AVAL	00000100	-14	29.11 29.23
AVALHI	01000000	-16	
EVAL	External	28	29.27 29.34
RVAL1	SEC1:00000000	-21	
RVAL2	SEC2:00000000	-24	

Figure 8-1. Sample Assembly Listing (cont.)

## The Object Module

During pass two of processing, the assembler produces a relocatable object module. The object module is a file that consists of variable length records in the binary IEEE-695 format. The object module contains specifications for loading the memory of the target microprocessor and provides the necessary linkage information required to link object modules together.

Relocatable object modules can be loaded into the linker and converted to a single absolute program in one of three different formats (Motorola S-record, IEEE-695, or HP-OMF). The absolute object module can then be loaded into a development system, used to program a PROM, or read into the simulator.



# Linker Operation 9

---

## Introduction

Many programs are too long to conveniently assemble as a single module. 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 LNK68K Linker.

The primary functions of the linker are to:

1. Resolve external references between modules and check for undefined references (linking)
2. Adjust all relocatable addresses to the proper absolute addresses (loading)
3. Output the final absolute object module(s)

After the separate program modules are linked and loaded, the output module functions as if it had been generated by a single assembly.

When an absolute load is performed, relocatable addresses are transformed into absolute addresses, external references between modules are resolved, and the final absolute symbol value is substituted for each external symbol reference. The linker lets you specify program section addresses and external definitions. It also lets you assign the final load address and section loading order. Absolute output is produced in Motorola hexadecimal S-record format, IEEE-695 format (default), or in Hewlett-Packard HP-OMF format.

LNK68K can combine relocatable object modules into a single relocatable object module suitable for later relinking with other modules. This feature is known as incremental linking. In an incremental link, external references are resolved whenever possible. No section address assignment or resolution of relocatable references is performed.

## Linker Features

The LNK68K Linker supports the following features:

- Absolute output in Motorola hexadecimal S-record format, in IEEE-695 absolute format (default), or in Hewlett Packard HP-OMF format
- Independently-specified relocatable section load addresses
- Specified relocatable section loading order
- Support for incremental linking, including library modules
- Definition of external symbols at link time
- Changing values of previously defined externals at link time
- Loading of object modules from a library (The linker includes only those modules from a library that are necessary to resolve external references)
- Inclusion of symbols and line number information in the absolute object module for symbolic debugging
- Cross-reference table generation
- Scatter loading
- First-fit algorithm placement of object modules memory for efficient memory utilization
- Complex relocation
- A2-A5 address register-relative addressing
- Data initialization from ROM and other locations
- Automatic copying of initialized data values, which can be placed in ROM

## Program Sections

For effective use of the ASM68K Assembler and LNK68K Linker, you must understand sections and their various section attributes.

A section is a region of memory that contains information. There can be up to 30,000 general purpose sections which can contain both instructions and/or data. Each section contains its own location counter and typically is a logically distinct part of the total program. Instructions in one section can make a reference to any other sections.

Sections are defined by the following attributes:

- Each section is identified by a symbolic name.
- A section is absolute or relocatable.
- A section has a type.
- The components of a section can be aligned to a particular byte spacing.

Different relocatable module sections (or subsections) with the same name are combined into the same section unless linker commands split the section. The combined section refers to the total code from all object modules which is associated with the section name. Absolute sections are not combined.

The subsections of a section are loaded into a contiguous block of memory and do not overlap. The size of a section is the sum of the sizes of all its subsections. Sections have a type attribute and an alignment attribute.

## Absolute Sections

The absolute section is that part of the assembly program that contains no relocatable information but is to be loaded at fixed locations in the user's memory. Absolute code is placed into the output module exactly where specified by the input object modules. Note that only those bytes of memory which actually contain data are considered to be within the absolute section. Areas skipped by a DS directive, for instance, are not stored as data. However, the section size will reflect the area reserved by the DS directive.

Absolute sections have a predefined load address. Absolute code is placed into the output module exactly where specified by the input object modules. An absolute section contains no relocatable information.

## Relocatable Sections

A relocatable section is one of up to 30,000 general purpose sections which can contain both instructions and/or data. Typically, each section is a logically distinct part of the total program. Instructions in one section can make references to other sections.

Each section is identified by a symbolic name. The same section name can appear in different relocatable object modules. The section, as a whole, refers to code from all object modules which are associated with the section name. 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. You can override the default for combining sections by the linker with the **MERGE** or **ALIAS** linker command.

In the assembler, sections can be given numbers rather than names. However, the assembler translates such numbers into names as described in the chapter called *Relocation*. From the linker's point of view, all sections are named.

Each relocatable section has four attributes:

- Common/noncommon
- Short/long
- Section type
- Section alignment

Sections containing these types of attributes are described below. For more information on relocatable modules using these attributes, refer to Chapter 4, *Relocation*.

### **Common Section**

A common section contains variables that can 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 in the same common section, the output module will contain all such code even though it can overlap. The reactions of various programs which read the output module to this condition are not predictable. For this reason, a warning flag is given at assembly time to any code produced in a common section.

### **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 plus alignment.

### **Short Section**

A short section can be referenced by the absolute short addressing mode and, for this reason, it must be loaded into the areas of memory which can be reached by a 16-bit sign-extended address (see Table 9-1). The target chip can be specified by the **CHIP** command. The linker 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 linker commands. The default address width may be changed using the **CHIP** directive in linker commands.

## Long Section

A long section is a section which is not short and which can be placed anywhere in memory.

**Table 9-1. 16-Bit Absolute Address Memory Areas**

Chip	Address Range
68000/10, 68302, 68EC000, 68HC000/01	0 to \$7FFF \$FF8000 to \$FFFFFF
68008	0 to \$7FFF \$F8000 to \$FFFFFF
68020/30/40, 680330/331/332/333/340, 68EC020/030/040, CPU32	0 to \$7FFF \$FFFF8000 to \$FFFFFF

## Section Type

There are four types of relocatable sections:

1. **C** — Program Code
2. **D** — Data
3. **M** — Mixed code and data
4. **R** — ROMable Data

The section type attribute has two purposes. First, it can serve as documentation of what a section contains. Program code sections generally contain instructions. Data sections generally contain read/write data items. ROMable data sections generally contain read-only data items.

The section type attribute affects the production of HP-OMF symbolic information in the **asmb\_sym** (assembler symbol) and **link\_sym** (linker symbol) files. The HP-OMF 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**.

## Section Alignment

The beginning address of each file's contribution to a section is aligned by default at word boundaries. This alignment can be increased by using the **ALIGN**, **ALIGNMOD**, and **PAGE** commands. Specified absolute addresses will be

rounded up to the next word boundary if they do not fall on one. For more information about section alignment, refer to Chapter 4, *Relocation*.

The section alignment attribute can be any power of 2. The section alignment attribute affects the beginning address of each file's contribution to a section (i.e., subsection). That is, if several files each define a relocatable section named A and the alignment for this section is 4, then the beginning address of section A in each file will be rounded up to a modulo 4 boundary if necessary. For more information on section alignment, see the **ALIGN** linker command.

## Section Type

A section can have one of the types listed in Table 9-2.

Table 9-2. Section Types

Section Type	Meaning
C	Program code
D	Data
M	Mixed code and data
R	ROMable data

The section type attribute has two purposes. First, it can serve as documentation of what a section contains. Program code sections generally contain instructions. Data sections generally contain read/write data items. ROMable data sections generally contain read-only data items.

The section type attribute affects the production of HP-OMF symbolic information in the **asmb\_sym** (assembler symbol) and **link\_sym** (linker symbol) files. The HP-OMF 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

You can control both the order in which sections are assigned space in memory and the initial address (load address) of any or all sections. Specifying the load address of a section does not alter the order in which sections are assigned space, but it affects the location in memory of subsequent sections that do not have load addresses specified.

Several different kinds of addresses are used 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** 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.
2. Allocate short sections.
3. Allocate long sections.

The order in which short or long sections are assigned memory is as follows:

1. Any sections listed in **ORDER** commands (for long sections) or **SORDER** commands (for short sections) in the sequence in which they were named in that command.
2. Any other sections belonging to the group in the sequence in which their names were encountered by the linker.

The linker encounters a section name when the name appears in a command, or when a module which refers to that section name is loaded. Section names appear in relocatable object modules produced by the assembler in the sequence in which they appeared in directives in the assembler source input. You can use the **MERGE** command to override the default combining of sections by common name.

#### Note

Library relocatable object modules which are not selected for inclusion in the absolute object module do not have their section names examined by the linker.

To assign memory to a section, you need to assign it a load address. For those sections whose load addresses are specified in a **SECT** or **COMMON** directive, nothing more need be done. Otherwise, the following will occur:

1. The first short section is loaded at the first available space after the base address as specified by the **BASE** command. If no **BASE** command is given, the default base address is 0.
2. 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 linker will not split a short section between low and high memory.
3. The first long section is loaded immediately above the short section that is 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. The first long section will be loaded at the base address as specified in item 1 above if there were no short sections.
4. Subsequent long sections start immediately above the preceding long section.

## Relocation Types

By default, sections are word relocatable. That is, they must begin on an even address. If an odd load address is specified, it will be rounded up. You can, however, use the **PAGE** and **CPAGE** commands to specify that certain sections are page relocatable, meaning that their starting address is rounded up to a multiple of \$100. This page relocatability can be turned on and off between modules, which lets you control the relocation type of each subsection.

Page relocation is useful for debugging since it means the absolute addresses assigned by the linker will match the last two digits of the relocatable addresses shown on the assembler listing.

In the typical load sequence, the linker places contiguously in memory all subsections of the first section it assigns, 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 linker will start that subsection at a page boundary to preserve relocation. Whenever any subsection of a section is page relocatable, the first subsection in that section starts on a page boundary, unless the section has a load address specified. To avoid wasting memory, you can always specify section load addresses. If paging is in effect at the time the

first subsection of a section is loaded with the **LOAD** command, even a specified load address will be rounded up.

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.

## Incremental Linking

The linker can produce a single relocatable object module from assembled relocatable object modules through the process of incremental linking. The linker resolves all external references between the modules loaded, and allows undefined external references to other modules to exist in the output object module. The external references are reported on the link map.

Incremental linking is useful because it lets groups of users easily share relocatable object modules for the joint development of code. Long lists of previously checked object modules do not have to be linked with those modules currently under development. Only one incrementally linked module has to be linked, making it unnecessary to know all the original module names that are being linked with the new code.

The following example shows incremental linking of four test modules: TEST1, TEST2, TEST3, and TEST4. In the first part of the example, a normal load is performed requiring four load commands. In the second part of the example, three of the modules are incrementally linked into TEST123. Then, TEST123 and TEST4 are linked to produce one absolute output file.

### Example:

```
TEST.CMD      * A command file that consists of four
               * load commands to link four relocatable
               * object modules.
LOAD TEST1
LOAD TEST2
LOAD TEST3
LOAD TEST4
```

The output object module produced is TEST.ABS.

The same result could be achieved with the two following load command sequences:

TEST123.CMD

- \* A command file consisting of three
- \* load commands to link three
- \* relocatable object modules.

LOAD TEST1

LOAD TEST2

LOAD TEST3

The output object module produced is TEST123.OBJ.

TESTF.CMD

- \* A command file consisting of two
- \* load commands to link two
- \* relocatable object modules, one of
- \* which is a combination of
- \* previously linked modules.

LOAD TEST123.OBJ

LOAD TEST4

The output object module produced is TESTF.ABS.

## Symbols in Linker Commands

### Program Identifiers

In linker commands, names that represent program identifiers (for example, the identifier in the **PUBLIC** command) must conform to the assembly language restrictions for identifiers, which are:

1. The first character must be one of the following:

a-z    A-Z    ?    @    \_    \$

2. Subsequent characters may be any of the above, plus:

0-9

3. The maximum symbol length is 127 characters. Symbols are truncated to 15 characters (on output, not internally) if **FORMAT HP** applies.

4. In module and segment names, upper- and lower-case letters are not distinct. For all other names, they are distinct unless the **CASE** command is used.

**Example:**

```
PUBLIC NEWNAME 100H  
public newname 100h
```

In the above example, **NEWNAME** and **newname** refer to different identifiers. The linker command **PUBLIC** is recognized in both upper- and lower-case.

If you run the assembler without **CASE**, the assembler will remap all symbols to uppercase. Given this, the linker commands naming symbols must use uppercase names.

Names originating from the Microtec Research C Compiler retain their original capitalization. Segment names are still uppercase.

## Continuation and Escape

The continuation character is the pound sign (#). The linker treats the continuation character and all characters between it and the End-Of-Line as if they were a single blank, followed by the first character of the next line.

The continuation character is a token separator. Thus, a single token cannot be continued from one line to the next, and a token cannot contain an ampersand without using the escape character, backquote (`).



# Linker Commands 10

---

## Introduction

The LNK68K Linker reads a sequence of commands from a specified command file and/or an interpreted command line. This chapter provides reference information about the available commands and their syntax. For more information on command line and command file usage, see the *Microtec Research ASM68K User's Guide*.

## Command Syntax

Commands and command arguments can begin in any column. Command arguments must be separated from the command by at least one blank. Only one command is permitted per line. Commands can be continued on multiple lines using a pound sign (#) as the last character of the line to be continued. Comments are indicated by an asterisk.

Table 10-1 shows how numeric command arguments can be represented.

**Table 10-1. Representation of Numeric Command Arguments**

Type	Indicator
Hexadecimal	Preceded by \$ or terminated with H or X
Binary	Terminated with B
Octal	Terminated with O
Decimal	Terminates with D

If a numeric command argument is not explicitly described using these methods, the linker assumes the argument to be decimal. Using two hexadecimal indicators at the same time is illegal.

**Example:**

The following command file shows numeric command arguments.

```
sect sect1 = 1010B ; binary      (A hex)
sect sect2 = 1010O ; octal       (208 hex)
sect sect3 = 1010D ; decimal     (3F2 hex)
sect sect4 = 1010   ; decimal     (3F2 hex)
sect sect5 = 1010H ; hexadecimal (1010 hex)
sect sect6 = $1000 ; hexadecimal (1000 hex)

        load lod_expres
end
```

The resulting module summary will be as follows:

```
        .
        .
MODULE SUMMARY
-----
MODULE      SECTION:START    SECTION:END      FILE
lod_expres  sect1:0000000A    sect1:0000000D  lod_expres.o
            sect2:00000208    sect2:0000020B
            sect3:000003F2    sect3:000003F5
            sect4:000003F2    sect4:000003F5
            sect5:00001010    sect5:00001013
            sect6:00001000    sect6:00001003
        .
        .
```

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 underscore (\_). Subsequent characters can be a dollar sign (\$) or a decimal digit.

## Case Sensitivity

Section names, symbols, and reserved names (e.g., IEEE) are by default case-sensitive in the linker command file. The **CASE** linker command can alter this default. The **OPT CASE** assembler directive specifies that symbols are case-sensitive in the assembler.

## Command Position Dependencies

The linker will process commands in the following order and also handle positional dependencies by the following rules:

1. Preprocessed commands such as the **INCLUDE** command are expanded before any linker commands are processed.
2. Nonposition dependent commands are processed next:

ALIGN  
ALIGNMOD  
BASE  
CHIP  
ERROR  
FORMAT  
NAME  
START  
LISTMAP  
LISTABS  
RESADD  
RESMEM  
SECTSIZE  
WARN

3. 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
SYMTRAN	should be before any command using names
[NO]PAGE	should be before any command using names
CPAGE	
[NO]DEBUG_SYMBOLS	
LOAD	
LOAD_SYMBOLS	
EXTERN	
PUBLIC	

4. Commands that are position-independent in the command file are processed next, but they are operated on in the following order:

1. COMMON, SECT
2. MERGE
3. ALIAS

4. ORDER, SORDER
5. ABSOLUTE, INDEX, INITDATA

6. Commands that end command processing are processed last:

END  
EXIT

## Linker Commands

The linker commands in this chapter are organized alphabetically. An alphabetical listing of the linker commands is shown in Table 10-2. The commands are described in the pages following the table.

**Table 10-2. Linker Commands**

Command	Function
ABSOLUTE	Specifies Sections to Include in Absolute File
ALIAS	Specifies Section Assumed Name
ALIGN	Sets Alignment for Named Section
ALIGNMOD	Sets Alignment for Module Sections
BASE	Specifies Location to Begin Loading
CASE	Controls Case-Sensitivity
CHIP	Specifies Target Microprocessor
Comment	Specifies Linker Comment
COMMON	Sets Common Section Load Address
CPAGE	Sets Common Section to be Page Relocatable
DEBUG_SYMBOLS	Retains or Discards Internal Symbols
END	Ends Command Stream and Finishes Linking
ERROR	Modifies Message Severity
EXIT	Exits Linker Without Linking
EXTERN	Creates External References
FORMAT	Selects Output Format
INCLUDE	Includes a Command File
INDEX	Specifies Run-Time Value of Register A <sub>n</sub>
INITDATA	Specifies Initialized Data in ROM
LISTABS	Lists Symbols to Output Object Module
LISTMAP	Specifies Layout and Content of the Map

(cont.)

**Table 10-2. Linker Commands (cont.)**

<b>Command</b>	<b>Function</b>
LOAD	Loads Object Modules and/or Library Modules
LOAD_SYMBOLS	Loads Symbol Information of Specified Object Modules
LOWERCASE	Shifts Names to Lowercase
MERGE	Combines Named Module Sections
NAME	Specifies Output Module Name
ORDER	Specifies Long Section Order
PAGE	Sets Page Alignment
PUBLIC	Specifies Public Symbols (External Definitions)
RESADD	Reserves Regions of Memory
RESMEM	Reserves Regions of Memory
SECT	Sets Section Load Address
SECTSIZE	Sets Minimum Section Size
SORDER	Specifies Short Section Order
START	Specifies Output Module Starting Address
SYMTRAN	Transforms Public/External Symbols
UPPERCASE	Shifts Names to Uppercase
WARN	Modifies Message Severity

## ABSOLUTE — Specifies Sections to Include in Absolute File

### Syntax

```
ABSOLUTE sname [, sname] . . .
```

### Description

<i>sname</i>	Names relocatable section to be placed into the absolute output object module.
--------------	--

The **ABSOLUTE** command lets you specify that only the code and data from certain, specified program sections be included in the absolute output file. Without the **ABSOLUTE** command, code and data from all sections in the input modules are placed into the absolute output file.

The **ABSOLUTE** command allows implementation of code overlays. Typically, in an application employing overlays, there is a main code section and several overlay sections. Usually, the main section resides in memory. The overlays are not resident, but are loaded into memory as needed during program execution. However, the code in the overlay sections needs to be linked to locations in the main section and vice versa.

When using the **ABSOLUTE** command, only code and data from relocatable sections are put into the output. Code and data from absolute sections (i.e., specified using **ORG**) are never put into the output when the **ABSOLUTE** command is used.

### Example

The following example shows how to link an application consisting of a main program and two overlays. It requires three load operations and three linker command files.

All the code and data for the main section is in section **MAINSECT**. All the code for the 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 the output file.  
LOAD   MOD1,MOD2,...,MODn    * Load all modules for main,  
                             * overlay 1, and overlay 2.  
END
```

## Linker commands 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 the output file.  
LOAD      MOD1,MOD2,...,MODn    * Load all modules for main,  
                             * overlay 1, and overlay 2.  
END
```

## Linker commands 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, and overlay 2.  
END
```

## ALIAS — Specifies Section Assumed Name

### Syntax

```
ALIAS alias_sname, sname
```

### Description

*alias\_sname*      Specifies assumed section name.

*sname*      Specifies section name.

The **ALIAS** command lets you specify that *sname* be considered the same as *alias\_sname*. The linker will load parts of *alias\_sname* sections contiguously as if they were parts of the same *sname* section. The resulting output object file will show the two combined sections under the name *alias\_sname*. Without the **ALIAS** command, the linker 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 combine only two sections, and the subsections of the two sections are combined in the order in which they are encountered with the **LOAD** command. **MERGE** command arguments on the other hand are combined in the order specified. It is an error to use **ALIAS** and **MERGE** during the same link session.

### Notes

**MERGE** and **ALIAS** are mutually exclusive. Any attempt to use both commands in the same session will result in an error.

### Example

The following command file shows the **ALIAS** command.

```
ALIAS sect1 sect3
FORMAT IEEE

LOAD alias
END

OUTPUT MODULE NAME:    alias
OUTPUT MODULE FORMAT:  IEEE
```

The resulting link map shows that sect3 is considered the same as sect1.

SECTION SUMMARY

---

SECTION	ATTRIBUTE	START	END	LENGTH	ALIGN
sect1	NORMAL CODE	00000000	00000007	00000008	2 (WORD)
sect2	NORMAL CODE	00000008	0000000B	00000004	2 (WORD)
sect4	NORMAL CODE	0000000C	0000000F	00000004	2 (WORD)
sect3	(sect1)	00000000	00000007	00000008	2 (WORD)

MODULE SUMMARY

---

MODULE	SECTION:START	SECTION:END	FILE
alias	sect1:00000000	sect1:00000003	alias.o
	sect2:00000008	sect2:0000000B	
	sect1:00000004	sect1:00000007	
	sect4:0000000C	sect4:0000000F	

## ALIGN — Sets Alignment for Named Section

### Syntax

```
ALIGN sname=align_value
```

### Description

*sname*              Specifies section name.

*align\_value*        Specifies a constant which is a power of 2 between 1 and  $2^{32}$ .

Every relocatable module section has an alignment attribute. When the module section is located, the linker makes its base address a multiple of the alignment.

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.

### Example

If **modulea.src** has the following:

```
.option        case
.section       sect1,data
.datab.b      1,3
.section       sect2,data,align=4
.datab.b      1,4
.section       sect3,text,align=1
nop
nop
.datab.b      1,5
.end
```

and **moduleb.src** has the following:

```
.option        case
.section       sect1,data
.datab.b      1,3
.section       sect2,data,align=1
.datab.b      1,4
.section       sect3,text,align=4
nop
nop
.datab.b      1,5
.end
```

and the linker command file, **lnk\_align.cmd**, contains the following:

```
ALIGN sect1,512      * align sect1 combined section on
                     * 512 byte boundary
ALIGNMOD sect1,16    * align sect1 module sections on
                     * 16 byte boundaries
ALIGNMOD sect2,8     * align sect2 module sections on
                     * 8 byte boundaries
ORDER sect3,sect2,sect1
LOAD modulea,moduleb
END
```

Once the two assembler files are assembled with the command line:

```
mcc68k -c lnk_align.cmd -m > mod.map
```

the resulting link map includes the following:

```
OUTPUT MODULE NAME:      lnk_align
OUTPUT MODULE FORMAT:   IEEE
OUTPUT MODULE TYPE:     SMALL
```

#### SECTION SUMMARY

SECTION	ATTRIBUTE	START	END	LENGTH	ALIGN
sect3	NORMAL CODE	00000000	00000006	00000007	16
sect2	NORMAL DATA	00000008	00000010	00000009	8
sect1	NORMAL DATA	00000200	00000210	00000011	512

#### MODULE SUMMARY

MODULE	SECTION:START	SECTION:END	FILE
modulea	sect1:00000200 sect2:00000008 sect3:00000000	sect1:00000200 sect2:00000008 sect3:00000002	modulea.o
moduleb	sect1:00000210 sect2:00000010 sect3:00000004	sect1:00000210 sect2:00000010 sect3:00000006	moduleb.o

START ADDRESS: 00000000

Link Completed

## ALIGNMOD — Sets Alignment for Module Sections

### Syntax

```
ALIGNMOD sname=align_value
```

### Description

*sname*              Specifies section name.

*align\_value*        Specifies a constant which is a power of 2 between 1 and  $2^{32}$ .

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.

## BASE — Specifies Location to Begin Loading

### Syntax

```
BASE number
```

### Description

<i>number</i>	Specifies an absolute number.
---------------	-------------------------------

The BASE command specifies the base address of the output object module. The base address is the lowest address at which the first section in a module is placed, provided that this section does not have its load address specified in a SECT or COMMON command.

### Example

The following command file does not have a BASE command.

```
LOAD base  
END
```

The resulting link map shows the default base address of the first section is 0.

---

SECTION SUMMARY

---

SECTION	ATTRIBUTE	START	END	LENGTH	ALIGN
sect1	NORMAL CODE	00000000	00000003	00000004	2 (WORD)
sect2	NORMAL CODE	00000004	00000007	00000004	2 (WORD)
sect3	NORMAL CODE	00000008	0000000B	00000004	2 (WORD)
sect4	NORMAL CODE	0000000C	0000000F	00000004	2 (WORD)
...					

The following command file has a BASE command.

```
BASE $1400  
LOAD base  
END
```

The resulting link map shows the base address of the first section is 1400 hexadecimal.

## SECTION SUMMARY

---

SECTION	ATTRIBUTE	START	END	LENGTH	ALIGN
sect1	NORMAL CODE	00001400	00001403	00000004	2 (WORD)
sect2	NORMAL CODE	00001404	00001407	00000004	2 (WORD)
sect3	NORMAL CODE	00001408	0000140B	00000004	2 (WORD)
sect4	NORMAL CODE	0000140C	0000140F	00000004	2 (WORD)

---

## CASE — Controls Case-Sensitivity

### Syntax

[NO]CASE [*class* [, *class*] ...]

### Description

<i>class</i>	Specifies one of the following class names:
PUBLICS	All public and external names.
MODULES	All section names.
SECTIONS	All module names.

The CASE command controls the case-sensitivity of various classes of symbolic names by specifying that upper- and lower-case characters are distinct in name comparisons. Symbolic names in the indicated by *class* are not modified on input.

NOCASE forces upper-case names.

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).

### Notes

Related commands include LOWERCASE and UPPERCASE.

The CASE command takes immediate effect and should be used early in the command file.

### Example

Given the following command file:

```
CASE      PUBLICS
LISTMAP   PUBLICS
LOAD      modulea, moduleb, modulec
END
```

All public and external names will be unchanged in the linker's output file. The generated link map will contain a PUBLIC SYMBOL TABLE section that will show all the public and external names:

PUBLIC SYMBOL TABLE

---

SYMBOL	SECTION	ADDRESS	MODULE
G1	sect3	00001200	modulea
G2	sect3	00001204	moduleb
G3	sect3	00001208	modulec

## CHIP — Specifies Target Microprocessor

### Syntax

```
CHIP target[,n]
```

### Description

*target*

Specifies target microprocessor. Valid values are: 68000, 68008, 68010, CPU32, 68020, 68030, 68040, 68HC000, 68HC001, 68EC000, 68330, 68331, 68332, 68333, 68340, 68EC020, 68EC030, or 68EC040.  
(default: 68000)

*n*

Overrides the default target bus width. The maximum address is  $2^n - 1$ . The high short-addressable area address range is from ( $2^n - \$8000$ ) to ( $2^n - 1$ ).  
(default: maximum width for the target microprocessor)

The CHIP command stipulates on which microprocessor the linked code is to run. This command affects the section alignment attribute (see *Section Alignment Attribute* in Chapter 4, *Relocation*, in this manual).

The chief differences among the microprocessors in the 68000 family pertain to the size of the address space and to the addresses of the high memory area which can be accessed with the absolute short addressing mode. The linker places sections with the short attribute in this area of memory (or in the low short-addressable area of memory, which is from 0 to \$7FFF for all targets).

If a CHIP command is not present, the target microprocessor type defaults to the largest model used in the modules. For example, if three modules assembled on the 68000, the 68010, and the 68020 are ready to be linked and no CHIP command is indicated before the LOAD command, the target microprocessor will be 68020.

All absolute addresses which appear in later commands or object modules are checked against the bounds established by the CHIP command.

The differences in memory addressing among the 68000 family chips are summarized in Table 10-3.

Table 10-3. 68000 Family Memory Addressing

CHIP	Maximum Address	Bits	High Short Addressable Area of Memory
68000	\$FFFFFF	24	\$FF8000 to \$FFFFFF
68302	\$FFFFFF	24	\$FF8000 to \$FFFFFF
68EC000	\$FFFFFF	24	\$FF8000 to \$FFFFFF
68HC000	\$FFFFFF	24	\$FF8000 to \$FFFFFF
68HC0001	\$FFFFFF	24	\$FF8000 to \$FFFFFF
68008	\$FFFFF	20	\$F8000 to \$FFFFF
68010	\$FFFFFF	24	\$FF8000 to \$FFFFFF
68020	\$FFFFFFFFF	32	\$FFFF8000 to \$FFFFFFFFF
68030	\$FFFFFFFFF	32	\$FFFF8000 to \$FFFFFFFFF
68040	\$FFFFFFFFF	32	\$FFFF8000 to \$FFFFFFFFF
68EC020	\$FFFFFFFFF	32	\$FFFF8000 to \$FFFFFFFFF
68EC030	\$FFFFFFFFF	32	\$FFFF8000 to \$FFFFFFFFF
68EC040	\$FFFFFFFFF	32	\$FFFF8000 to \$FFFFFFFFF
68330	\$FFFFFFFFF	32	\$FFFF8000 to \$FFFFFFFFF
68331	\$FFFFFFFFF	32	\$FFFF8000 to \$FFFFFFFFF
68332	\$FFFFFFFFF	32	\$FFFF8000 to \$FFFFFFFFF
68333	\$FFFFFFFFF	32	\$FFFF8000 to \$FFFFFFFFF
68340	\$FFFFFFFFF	32	\$FFFF8000 to \$FFFFFFFFF
CPU32	\$FFFFFFFFF	32	\$FFFF8000 to \$FFFFFFFFF

### Notes

Overriding the maximum address bus width implied for the target microprocessor lets the maximum address in memory be limited regardless of the maximum address bus width possible for the chip.

### Example

```
FORMAT S
CHIP 68020,24
```

For S-record output, the target address bus width argument sets the S-record type. By default, CHIP 68020 causes S3 records to be produced. However, if the bus width is 24 or less, S2 records are produced. If the address bus width is greater than 24, S3 records are produced.

**Example**

```
* This is the linker command file with
* the default bus width picked for the 68020.
*
CHIP 68020
FORMAT S
LOAD test
END
```

This linker command file generates an absolute file with S3 records (default):

```
S00600004844521B
S306000000006495
S5030001FB
S705000000000FA
```

```
* This is the linker command file with
* the bus width, 20, explicitly indicating
* that S2 records are desired.
*
CHIP 68020,20
LOAD test
END
```

This linker command file generates an absolute file with S2 records (since the bus width qualifier is 20):

```
S00600004844521B
S2050000006496
S5030001FB
S804000000FB
```

## Comment — Specifies Linker Comment

### Syntax

\**comment*  
or  
;*comment*

### Description

Comments document linker command sequences. If an asterisk (\*) is used to indicate the start of a comment, it must be located in column 1 or preceded by blanks or tabs. If a semicolon (;) is used, it can occur anywhere on the line.

The semicolon is allowed anywhere in the command file. The semicolon is even allowed on the same line as other commands.

### Example

```
* Linker example.  
*  
*  
LOAD mod1.obj ; This is the only module loaded.  
END
```

## COMMON — Sets Common Section Load Address

### Syntax

```
COMMON sname [= | ,] value
```

### Description

*sname*              Specifies the section name.

*value*              Specifies the load address of the common section.

The **COMMON** command specifies the load address of a common section. If **COMMON** is used, it must be specified before any **LOAD** command.

Enter the section name followed by the address at which to begin loading the section. The address specified is rounded up to the next higher word boundary in all cases and, if paging is specified for this common section, to the next higher page boundary.

If the section does not already exist, the **COMMON** command will create a new section with zero size. The type of the section created will be common.

If this is the first occurrence of this section name, it is given the attributes of common and long.

Multiple **COMMON** commands with the same section name are accepted without a warning, but only the last one is used.

The *value* is separated from the section name by a blank, comma, or equal sign.

### Example

The following command file sets the load address of the common section *sect1* at 2000 hexadecimal.

```
COMMON sect1 = $2000 ; Locate the common section sect1 at 2000 hex
LOAD common1, common2
END
```

The resulting link map shows that the common section *sect1* was loaded for *common1* and *common2* at 2000 hexadecimal.

```
...
OUTPUT MODULE NAME:    common
OUTPUT MODULE FORMAT: IEEE
```

## SECTION SUMMARY

SECTION	ATTRIBUTE	START	END	LENGTH	ALIGN
sect1	COMMON	00002000	000020FF	00000100	2 (WORD)
sect2	NORMAL CODE	00000000	00000007	00000008	2 (WORD)

## MODULE SUMMARY

MODULE	SECTION:START	SECTION:END	FILE
common1	sect1:00002000	sect1:0000205F	common1.o
	sect2:00000000	sect2:00000003	
common2	sect1:00002000	sect1:000020FF	common2.o
	sect2:00000004	sect2:00000007	

...  
...

## CPAGE — Sets Common Section to be Page Relocatable

### Syntax

```
CPAGE sname
```

### Description

*sname*                  Specifies a section name.

The CPAGE command modifies the relocation type of common sections in the input object modules to page. It lets you override the default relocation type of word for a common section. The PAGE command, on the other hand, sets the relocation type of a noncommon section to page.

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.

### Notes

Once page relocation is turned on for a common section, it cannot be turned off for later subsections of the section.

If *sname* is the first occurrence of the section name, it is assigned the attributes common and long.

### Example

In the following command file, the starting address of sect1 is page-aligned to 2100 hexadecimal.

```
COMMON sect1 = $2020 ; Locate the common section sect1 at 2020 hex
CPAGE sect1 ; aligns the common section sect1 at 2100 hex

* sect1 in common1.o has a size of $60
* sect1 in common2.o has a size of $100
* non-common section sect2 has a size of 4 in both modules

LOAD common1, common2
END
```

The resulting link map shows that the load address for sect1 is 2100 hexadecimal.

.....  
OUTPUT MODULE NAME: cpage  
OUTPUT MODULE FORMAT: IEEE

SECTION SUMMARY

---

SECTION	ATTRIBUTE	START	END	LENGTH	ALIGN
sect1	COMMON	00002100	000021FF	00000100	2 (WORD)
sect2	NORMAL CODE	00000000	00000007	00000008	2 (WORD)

MODULE SUMMARY

---

MODULE	SECTION:START	SECTION:END	FILE
common1	sect1:00002100	sect1:0000215F	common1.o
	sect2:00000000	sect2:00000003	
common2	sect1:00002100	sect1:000021FF	common2.o
	sect2:00000004	sect2:00000007	

.....

## DEBUG\_SYMBOLS — Retains or Discards Internal Symbols

### Syntax

[NO] DEBUG\_SYMBOLS

### Description

The **DEBUG\_SYMBOLS** command provides module-by-module control of internal (local) symbols.

The **DEBUG\_SYMBOLS** command includes internal symbols in the map file as well as in the output object module; **NODEBUG\_SYMBOLS** suppresses them.

The **DEBUG\_SYMBOLS** and **NODEBUG\_SYMBOLS** commands can appear anywhere in a linker command file. The command remains in effect until it is overridden by its complementary command or until the end of the command file.

### Notes

The **DEBUG\_SYMBOLS** command works in conjunction with the **LISTMAP** and **LISTABS** commands. If **LISTMAP NOINTERNAL**s has been specified, the **DEBUG\_SYMBOLS** command will have no effect on the map file. Similarly, if **LISTABS NOINTERNAL**s has been specified, the **DEBUG\_SYMBOLS** command will have no effect on the absolute file. For **DEBUG\_SYMBOLS** to take effect, **INTERNAL**s must be turned on by either the **LISTABS** or **LISTMAP** command.

## END — Ends Command Stream and Finishes Linking

### Syntax

```
END
```

### Description

The END command starts the final steps in the load process. After an END command is found, the linker completes the load, produces an output object module, and returns you to the host computer operating system.

### Example

```
* Load first two modules
LOAD modulea.obj,moduleb.obj
* Load last module
LOAD modulec.obj
END
```

This command file loads modulea.obj, moduleb.obj, and modulec.obj; the linker then produces an output object module and returns you to the host operating system.

## ERROR — Modifies Message Severity

### Syntax

```
[NO]ERROR number[,number]...
```

### Description

*number*              Specifies a message number.

The **ERROR** and **NOERROR** commands specify that the message number indicated by *number* is to be treated as an error or a nonerror.

These commands have a global effect from the point at which the linker processes the information contained in the command. A subsequent **ERROR** or **NOERROR** command overrides any values set by a previous one.

### Notes

Fatal errors and the errors or warnings that are generated as a result of a syntactically or semantically incorrect **ERROR** or **NOERROR** command cannot be overridden.

See the related command **WARN**.

### Examples

```
ERROR 349
```

Upgrades warning 349 to an error condition.

```
NOERROR 301
```

Turns off error condition 301.

## **EXIT — Exits Linker Without Linking**

### **Syntax**

EXIT

### **Description**

The EXIT command can be the last command in the command stream. This command prevents the final load from taking place but reads all object modules and commands, checking for errors. EXIT stops linker execution and does not generate an output object module.

### **Example**

EXIT

## EXTERN — Creates External References

### Syntax

```
EXTERN name[,name]...
```

### Description

*name*                  Indicates the symbolic name of an external reference.

The EXTERN command creates external references for the linker to resolve. You can create these external references by using the EXTERN command to add an entry to the linker's internal symbol table for each named symbol. Information indicating that these are external symbols to be resolved is inserted into the symbol table. If the symbol already exists in the symbol table either as an external reference or definition, the EXTERN command is ignored and a warning is issued.

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.

### Notes

The -u *name* command line option has the same effect as inserting an EXTERN command into the command file before the first LOAD command, if any.

### Example

```
EXTERN g1
LOAD module1.o,module2.o,textern.lib
END
```

In this example, the symbol g1 is not defined in either module1 and module2. So if a definition of g1 exists in textern.lib, the appropriate module that contains the definition will be force-loaded in order to resolve the external reference.

## FORMAT — Selects Output Format

### Syntax

```
FORMAT type
```

### Description

<i>type</i>	Specifies output format generated by the linker. Valid values are:
HP	HP-64000 object module format (HP-OMF)
IEEE	Specifies Microtec Research extended IEEE-695 format. (default)
NOABS	Produces no absolute file; however, internal processing will be carried out and a map file will be produced if requested. This option cannot be specified with the INCREMENTAL, LIMITED, or LTL option.
S	Indicates the Motorola S-record file format.
INCREMENTAL	Specifies incremental linking. (IEEE-695 format)

The **FORMAT** command lets you specify the output object module format. If an unsupported *type* specifier is encountered, an error or warning will be generated and the default output format is produced. If you do not specify a **FORMAT** command, the default output format is produced.

The **FORMAT** command has a global effect. If multiple **FORMAT** commands are encountered, a warning message will be generated and the format specified by the first **FORMAT** command will be used. If no **FORMAT** command is specified, the output will be produced in IEEE-695 format which is also the default format for incremental linking.

If NOABS is specified, no object file will be produced; however, internal processing will be carried out and a map file will be produced if requested. As with the other format type, NOABS cannot be used in conjunction with any other type. If this condition occurs, an error or warning will be issued and the **FORMAT** command will be ignored.

## INCLUDE — Includes a Command File

### Syntax

```
INCLUDE filename
```

### Description

*filename*      Specifies a file to be included in the linker command file.

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 (8 for DOS hosts).

### 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 (UNIX path name in this example) was added for readability.

## INDEX — Specifies Run-Time Value of Register An

### Syntax

```
INDEX ?REGn,sectname[,offset]
```

### Description

<i>REGn</i>	Specifies an address register: A2, A3, A4, or A5. These registers are normally used in relation to compiler code generation.
<i>sectname</i>	Specifies a relocatable section whose load address plus <i>offset</i> is equal to the run-time value of address register <i>REGn</i> .
<i>offset</i>	Specifies a number to be added to the load address of the relocatable section specified. The result is the run-time value of <i>REGn</i> .

The **INDEX** command informs the linker of the run-time value of an address register *An* (where *n* = 2, 3, 4, or 5), which are normally used in relation to compiler code generation. The value you associate with a particular *An* register will equal a relocatable section's load address plus an offset value.

A public symbol equal to the run-time value specified will be created in the form **?An**. With the **XREF** directive, you can declare this public symbol to be an external symbol in the assembly language source file. You can use this symbol to initialize the appropriate address register.

The linker must 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:

```
<relocatable_expression>(An) OR (<relocatable_expression>,An)
```

Operands with the syntax shown above will generate the Address Register Indirect with 16-bit displacement addressing mode. The relocatable expression in the syntax above is an effective address (i.e., the location to be accessed). For more information on the notation used to show this syntax, see any *Motorola User's Manual*.

The linker calculates the 16-bit displacement using the equations:

$$\begin{aligned}&<\text{effective\_address}\> = \text{An} + \text{disp} \\ &\text{displacement} = <\text{effective\_address}\> - \text{An} \\ &\text{displacement} = <\text{relocatable\_expression}\> - \text{An}\end{aligned}$$

The **INDEX** command makes *An* a known value which lets the linker calculate the displacement. If you do not use the **INDEX** command, the linker will calculate the displacement under the assumption that the run-time value of the address register is zero.

Other addressing modes which can contain relocatable expressions in conjunction with address register indirection are: 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 Chapter 3, *Instructions and Addressing Modes*, for more information on how the **INDEX** command can be used with array addressing for registers A2 through A5.

## INITDATA — Specifies Initialized Data in ROM

### Syntax

```
INITDATA      sectname [, sectname] ...
```

### Description

*sectname*      Specifies a section name.

The INITDATA command initializes data variables in RAM before a program is executed. This feature is intended for use with data which has an initialized value and thus needs to be stored in ROM. For these applications, it is often necessary to copy the initialized values for all variables from the ROM section to a RAM section where the variables will reside and be modified during execution time.

The INITDATA command causes the linker to automatically create a new data section ??INITDATA, and to fill the section with a copy of all the initialized data values contained in the sections named in the command string. If a section type is specified, then all sections of the specified type are copied into the INITDATA section. The INITDATA command is legal only during a final link.

A separate initialization routine, `initcopy()`, should be called at the beginning of runtime to invoke the copy function and hence to reinitialize data variables each time the program runs. This routine checks that the special bytes generated by the linker in the section ??INITDATA provide the necessary information such as the copy destination address, copy size, and the mark for the end of the section content. The `initcopy()` routine is provided in the run-time library with the compiler distribution files.

The ROM section ??INITDATA may be ordered and assigned an address using standard linker commands. The user cannot create this section from the assembler or compiler (unless it is an empty section representing a reference to the section, i.e., the .STARTOF. or .SIZEOF. operator was used).

### Examples

```
INITDATA      vars, strings
```

The following linker command file locates section data at \$8000 while specifying its ROM address at \$2000, following the prog section.

```
SECT    prog=$1000      ; Locate program in ROM.  
SECT    data=$8000      ; Locate data in RAM.  
INITDATA data           ; Generate the ??INITDATA  
                        ; segment with initial values.  
ORDER   prog,??INITDATA ; Put a copy of initial values  
                        ; in ROM to be copied into RAM  
                        ; on startup.  
LOAD    initest  
END
```

## LISTABS — Lists Symbols to Output Object Module

### Syntax

```
LISTABS option [, option] ...
```

### Description

*option*

One of the following:

[NO] INTERNALS Places the 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 for formats other than S-record.  
(default: INTERNALS, NOINTERNAL for S-record)

[NO] PUBLICS Places globally defined symbols into the output object module.  
(default: NOPUBLICS)

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.

### Notes

Since **LISTABS** command options have a global effect, options that are inconsistent with a previous **LISTABS** command cannot be specified in a succeeding **LISTABS** command (e.g., **LISTABS PUBLICS** can be followed by **LISTABS INTERNALS, PUBLICS**, but not by **LISTABS NOPUBLICS**.) If such a condition occurs, a warning message will be issued at this and at each subsequent point of conflict, and the first specification will be used.

## LISTMAP — Specifies Layout and Content of the Map

### Syntax

```
LISTMAP option [, option] ...
```

### Description

<i>option</i>	One of the following:
[NO]CROSSREF	Causes a cross-reference listing to be output to the map file. (default: NOCROSSREF)
[NO] INTERNALS [/BY_NAME   /NAME]	Causes a listing of the nonpublic (local) symbol table to be output to the map file. If /BY_NAME or /NAME is specified, the symbol table is listed in alphabetical order. (default: NOINTERNAL)
[NO] PUBLICS [/BY_ADDR   /ADDR   /BY_NAME   /NAME]	Causes a listing of the public symbol table to be output to the map file. If /BY_NAME or /NAME is specified, the public symbol table is listed in alphabetical order. If /BY_ADDR or /ADDR is specified, the public symbol table is listed in address order. (default: NOPUBLICS)
LENGTH <i>lval</i>	Specifies that the map file page length is set to <i>lval</i> lines where <i>lval</i> is a numeric value between 5 and 255. The default value is 55 lines including any header information generated by the linker. (default: LENGTH 55)

The LISTMAP command controls output to the linker's map file. The LISTMAP command options have a global effect. Multiple LISTMAP commands that do not have inconsistencies with previous LISTMAP commands can be specified and have a cumulative effect.

**Notes**

The **LISTMAP INTERNALS** command is affected by the **DEBUG\_SYMBOLS** command. Internal symbols loaded while **NODEBUG\_SYMBOLS** is in effect are not listed in the local symbol table.

If multiple **LISTMAP PUBLICS** commands appear, the last specification takes effect.

**LISTMAP INTERNALS** will cause slow linker execution. This option should only be used if debugging of local symbols is required.

## LOAD — Loads Object Modules and/or Library Modules

### Syntax

```
LOAD module [,module] ...
```

### Description

*module*

Specifies a file which contains the object module. Input object modules can consist of relocatable modules from the assembly process, relocatable modules from incremental linking, or libraries.

The **LOAD** command specifies one or more input object modules to be loaded or search for declarations of as yet unresolved externals. The linker differentiates between input modules on the basis of their internal format. When a library name is encountered, it will be searched only if unresolved externals remain and only those modules resolving externals will be loaded.

Libraries are searched in the order found. Therefore, backward references from one library to another will not be resolved. In this case, the name of a library will have to be specified again to resolve the remaining external references. For example, in the case where each of two libraries makes external references to each other, it is generally necessary to load one library twice to include all the necessary modules:

```
LOAD libA,libB,libA
```

Incremental linking accepts relocatable modules produced from the assembly and produces a single relocatable object module. See *Incremental Linking* in Chapter 9, *Linker Operation*, for more information.

Object modules can be read from a combination of files and 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 can use as many **LOAD** commands as needed.

The linker searches for *module* in the following locations and order:

1. Absolute path if specified.
2. Relative path.
3. Using the **MRI\_68K\_LIB** environment variable.

**Example**

```
* LOAD command example
BASE $4000
LOAD file1,file2
LOAD mathlib
END
```

## LOAD\_SYMBOLS — Loads Symbol Information of Specified Object Modules

### Syntax

```
LOAD_SYMBOLS module [, module] ...
```

### Description

*module*      Specifies the file in which an object module or library resides.

The LOAD\_SYMBOLS command allocates space for modules in the specified object modules and also retains all public symbol definitions. The allocated space is only for the use by linker for memory mapping and will not affect the object output file. Code and data for the specified modules are omitted from the output file; only symbol table and debug information is written to the absolute file.

Input object modules can consist of relocatable modules from the assembly process, relocatable modules from incremental linking, or libraries. The input file(s) specified, whether they contain relocatable object modules (output of the assembler) or libraries (output of the librarian), are differentiated on the basis of their internal format and are handled identically by the linker.

Incremental linking accepts relocatable modules produced from the assembly and produces a single relocatable object module. See *Incremental Linking* in Chapter 9 *Linker Operation*, for more information.

1. Absolute path (search will stop if an absolute path is specified and the file is not found)
2. Relative path
3. Using **MRI\_68K\_LIB** environment variable.

If *module* is a library, LOAD\_SYMBOLS retains all external symbols in each loaded module so that forward references within the library cause allocation of space for subsequent modules. If *module* is not a library, all external symbols are ignored.

### Example

```
LOAD_SYMBOLS mod1, mod2
```

## LOWERCASE — Shifts Names to Lowercase

### Syntax

```
LOWERCASE [class [, class] ...]
```

### Description

*class*              Specifies one of the following class names:

PUBLICS	All public and external names.
MODULES	All module names.
SECTIONS	All section names.

The **LOWERCASE** command causes the linker to shift names to lowercase on input. All symbolic names of the specified classes will appear in lowercase in the linker's output files.

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**).

### Notes

**LOWERCASE** takes immediate effect and should be used early in the command file.

See the related commands **CASE** and **UPPERCASE**.

### Example

Given the following command file:

```
LOWERCASE PUBLICS
LISTMAP  PUBLICS
LOAD     modulea, moduleb, modulec
END
```

All public and external names will be set to lowercase in the linker's output file. The generated link map will contain a PUBLIC SYMBOL TABLE section that will show all the lower-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

## MERGE — Combines Named Module Sections

### Syntax

```
MERGE merge_name merge_arg[,merge_arg]...
```

### Description

*merge\_name*      Specifies the name of the new, combined section.

*merge\_arg*      One of the following:

*sectname*      Section name.

{*sectname*,*module*}

Section name followed by a module  
name. The braces are required.

Both *sectname* and *module* can be replaced by the wild card  
character \* to indicate all sections or all modules

The **MERGE** command merges all the named subsections into a new section. The default linker behavior merges sections with the same name into a like-named section. This command lets you concatenate arbitrary lists of subsections. The new section can then be placed anywhere in memory using standard linker commands. If the section *merge\_name* already exists and it is not specified in the *merge\_arg*(s), the sections specified in *merge\_arg*(s) will be placed before *merge\_name* and the final combined sections will use the name *merge\_name*. Multiple **MERGE** commands with the same *merge\_name* are concatenated.

**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. See *Command Position Dependencies* at the beginning of this chapter for a description of the order in which commands are executed relative to other commands.

### Notes

**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,MOD1},{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/MOD1, 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 — Specifies Output Module Name

### Syntax

```
NAME name
```

### Description

*name*              Specifies the object module name. Any legal symbol can be used for the module name.

The **NAME** command specifies the name of the final output object module. In the standard Motorola S-record hexadecimal format, this appears on the first line of the output object file if the **LISTABS PUBLICS** linker command has been specified. Any symbol assigned values by the **PUBLIC** command at load time is considered to lie in this module which is defined at load time.

Any legal symbol can be used for the module name. If you do not specify a name, the name of the output module will be specified by the rules outlined in the *User's Guide*.

### Example

```
NAME READER
FORMAT S
LISTABS PUBLIC
LOAD mod1.obj,mod2.obj
END
```

## ORDER — Specifies Long Section Order

### Syntax

```
ORDER {sectname | (sect_type)} [, {sectname | (sect_type)}] ...
```

### Description

<i>sectname</i>	Specifies a section name.
<i>sect_type</i>	Specifies a section type enclosed in parentheses. The type can be one of the following:
C	Code
D	Data
M	Mixed
R	ROM data

The **ORDER** command changes the default order of assigning load addresses to sections. As described in Chapter 9, *Linker Operation*, the normal order of the sections is the order in which the linker encountered their names.

The **ORDER** command is designed for circumstances where 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 is of no particular importance. Remember, 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.

The **ORDER** command tells the linker to assign memory in the address order specified. The linker follows these rules for assigning memory space:

1. All absolute sections and those assigned addresses explicitly are placed in memory first.
2. The first section specified in the **ORDER** command is placed in memory at the first available location starting at the beginning of memory. The location is chosen with consideration for the alignment and size of the section.
3. All subsequent sections specified in the **ORDER** command(s) are placed in memory at the first available location (taking in consideration alignment and size of the section) after the previous **ORDER** section.
4. Sections named in the **ORDER** command with explicit addresses (rule #1) are checked to make sure that the specified order is maintained. The location of these sections will also set the starting point at which the linker

will begin looking for the starting point for next section named in the command.

5. If the order which the **ORDER** command specified cannot be maintained, the linker will issue a warning and continue trying to assign memory to the remaining sections in the **ORDER** command.
6. If a section named in the **ORDER** command is not found, the section is ignored and the linker will continue with the next named section, if any.

The **ORDER** command applies to long sections; the **SORDER** command applies to short sections. If a section name appears in these commands for the first time it is assigned the appropriate length attribute, but it is assigned neither the common nor the noncommon attribute so that subsequently it can be either.

If the name of a short section appears in the **ORDER** command, this is a fatal error. The final determination of which sections are short cannot be made until all modules have been read since any short module section declaration makes a section short.

If the name of a long section appears in the **SORDER** command, a warning is printed and the section is given the short attribute. This can occur if a **SECT**, **COMMON**, **PAGE**, **CPAGE**, or **NOPAGE** command precedes the **SORDER** command, since these commands assign newly-found sections the long attribute.

The same section name cannot appear twice in an **ORDER** command. Multiple **ORDER** commands are accepted without a warning. They are concatenated and must not contain duplicate names.

The order of sections within each group is specified by entering section names separated by commas. Any sections remaining within the group are assigned memory space after the sections specified in the command in the order in which their names were encountered by the linker.

An **ORDER** command can be continued to the next line by terminating it with one or more spaces followed by a pound sign (#). A continuation character must be placed like a comma between section names.

### Examples

```
ORDER SEC1, COMSEG  
SORDER SEC2, SHORTSEC
```

```
ORDER SECT1 #  
SECT2, SECT3 #  
SECT4
```

## PAGE — Sets Page Alignment

### Syntax

[NO] PAGE *section\_name*

### Description

*section\_name*      Specifies a section name.

The PAGE command modifies the relocation type of a section in the input object modules to "page." After the PAGE command is read, each subsection of the specified section loaded thereafter will be page relocatable until a NOPAGE command for the section is encountered. The NOPAGE command restores the relocation type of a section to word.

The PAGE command lets you override the default relocation type (i.e., word). After the PAGE command is read, each subsection of the specified section loaded thereafter will be page relocatable until a NOPAGE command for the section is encountered. The NOPAGE command is legal but unnecessary unless the specified section has previously appeared in a PAGE command.

### Notes

The PAGE command lets you begin each section on a page boundary for ease of debugging. After debugging is completed, delete the PAGE commands to avoid wasting memory space.

The NOPAGE command lets you turn off page relocation for modules which are already known to work correctly (libraries, for instance) in order to save memory space.

The NOPAGE command is legal but unnecessary unless the specified section has previously appeared in a PAGE command.

If PAGE is in effect and this is the first occurrence of some section name, it is given the attributes noncommon and long. Similarly, if NOPAGE is in effect, it is given the long attribute, but it is assigned neither the common nor the noncommon attribute.

**Example**

In the following command file, the page alignment is turned on, turned off, and then turned on for sect1.

```
SECT sect1 = $32 ; Locate the beginning of sect1 at 32 hex
PAGE sect1 ; Set page alignment for sect1 pushing the
             ; beginning of sect1 to 100 hex
LOAD page1
NOPAGE sect1 ; Turn off page alignment for sect1
LOAD page2
PAGE sect1 ; Reset to page alignment for sect1
LOAD page3
LOAD page4
END
```

The resulting link map shows the load order-dependent paging for sect1.

```
...
.
.
.
OUTPUT MODULE NAME:    page
OUTPUT MODULE FORMAT:  IEEE
```

**SECTION SUMMARY**

SECTION	ATTRIBUTE	START	END	LENGTH	AI TGN
sect1	NORMAL CODE	00000100	00000303	00000204	2 (WORD)

**MODULE SUMMARY**

MODULE	SECTION:START	SECTION:END	FILE
page1	sect1:00000100	sect1:00000103	page1.o
page2	sect1:00000104	sect1:00000107	page2.o
page3	sect1:00000200	sect1:00000203	page3.o
page4	sect1:00000300	sect1:00000303	page4.o

## PUBLIC — Specifies Public Symbols (External Definitions)

### Syntax

```
PUBLIC symbol1= {value | symbol2 [{+|-} offset]}
```

### Description

<i>symbol1</i>	Specifies a name for the public symbol being defined.
<i>value</i>	Specifies a constant number <i>value</i> to be assigned to the public symbol <i>symbol1</i> . <i>value</i> is treated as absolute and will not be relocated relative to any section base.
<i>symbol2</i>	Specifies another symbol whose value, relocation type, and section ID are assigned to <i>symbol1</i> .
<i>offset</i>	Specifies a constant number that is added or subtracted from <i>symbol2</i> .

The PUBLIC command defines and/or changes the value of an external definition (XREF). The external symbols are defined at load time, which may prevent reassembly. This command is position dependent.

Symbol names specified by the linker's PUBLIC command have 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 linker's symbol table along with the specified value and will then be available to satisfy external references from object modules.

Any *value* defined by this command is considered to be absolute (i.e., it does not lie in any relocatable section and will not be relocated relative to any section base).

### Notes

To ensure that all symbols have been defined, use the PUBLIC command immediately before the END command. Relative references are unchanged.

PUBLIC values are separated from symbols by blanks, commas, or an equal sign. If multiple PUBLIC commands having the same symbol are specified, the last value for that symbol holds and a warning message is issued.

Public symbols from libraries will not be used to resolve external references if the PUBLIC command defined the symbols before the library is loaded.

### Example

The following assembler file shows the public command.

```
Command line: a68k -l public
Line Address
1 ; Source file to demonstrate
2 ; PUBLIC command
3 ;
4 OPT CASE
5 XDEF sym1,sym2
6
7 ORG $1800
8 00001800 4E71 sym1: NOP
9 00001802 4E71 NOP
10
11 00005000 sym2 EQU $5000
12
13 END
```

The following command file shows the use of the PUBLIC command. The symbols sym1 and sym2 have the values \$100 and \$200, respectively.

```
LISTMAP PUBLICS
LISTABS PUBLICS
PUBLIC sym1 = $100
LOAD public
PUBLIC sym2 = $200
END
```

As shown by the PUBLIC SYMBOL TABLE, the symbols sym1 and sym2 are located at 100 and 200 hexadecimal, respectively.

.....  
.....  
MODULE SUMMARY  
-----

MODULE	SECTION:START	SECTION:END	FILE
public	:00001800	:00001803	public.o

PUBLIC SYMBOL TABLE  
-----

SYMBOL	SECTION	ADDRESS	MODULE
sym1		00000100	\$\$
sym2		00000200	\$\$
....			
....			

## RESADD — Reserves Regions of Memory

### Syntax

```
RESADD low_addr,high_addr
```

### Description

*low\_addr*      Specifies a starting address. This number is a numeric constant.

*high\_addr*      Specifies an ending address. This number is a numeric constant.

The RESADD command reserves specified memory locations from *low\_addr* to *high\_addr*. The reserved memory region is made into an absolute section that will show up in the SECTION SUMMARY of the link map.

### Notes

If a section overlaps a reserved region, a nonfatal error message is issued. The link 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*.

### Example

The command file:

```
RESNUM $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)

## RESMEM — Reserves Regions of Memory

### Syntax

```
RESMEM low_addr, size
```

### Description

*low\_addr*      Specifies a starting address. This number is a numeric constant.

*size*      Specifies the number of bytes to be reserved. This number is a numeric constant.

The RESMEM command reserves specified memory locations from *low\_addr* to *low\_addr+(size-1)*. The reserved memory region is made into an absolute section that will show up in the SECTION SUMMARY of the link map.

If a section overlaps a reserved region, a nonfatal error message is issued. The link will still continue to completion, but the resulting absolute file will contain sections at overlapping addresses.

### Example

The 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 — Sets Section Load Address

### Syntax

```
SECT section_name [, |= | ] value
```

### Description

*section\_name*      Specifies the section name.

*value*      Specifies the load address of the section.

The SECT command specifies the load address of a relocatable section. It must precede any LOAD commands.

If the section does not already exist, the SECT command will create a new section of zero size. The type of the section created will be noncommon.

The section name is entered, 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 specified either by the assembler object module or the linker ALIGN command and to the next page boundary if paging is in effect for the first subsection of the section.

A section may not exceed the  $2^{20}$  address space. If a section is set at a high address and its size causes it to exceed the address space, an error will be issued.

### Notes

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.

### Examples

```
SECT SECT1,$400  
SECT SECT2=$1320  
SECT SECT3 $1500
```

## SECTSIZE — Sets Minimum Section Size

### Syntax

```
SECTSIZE sname=size [, sname=size] ...
```

### Description

<i>sname</i>	Specifies a section name.
<i>size</i>	Specifies a constant representing the minimum section size, in bytes.

The SECTSIZE command specifies the minimum size in bytes of a combined continuous memory space defined by *sname*.

An error is generated if the section size defined is less than the size of the combined section unless the section is type COMMON. If the specified section does not exist, it will be created and considered to be non-COMMON.

An error is generated if the section size defined is less than the default size. An error is generated if the section does not exist.

### Example

```
SECTSIZE stack=$100
```

## SORDER — Specifies Short Section Order

### Syntax

```
SORDER [sectname | (sect_type)] [, [sectname | (sect_type)]] ...
```

### Description

<i>sectname</i>	Specifies a section name.
<i>sect_type</i>	Specifies a section type enclosed in parentheses. The type can be one of the following:
C	Code
D	Data
M	Mixed
R	ROM data

The **SORDER** command changes the default order of assigning load addresses to short sections (i.e., order in which the linker encountered their names).

The **SORDER** commands is designed for those who do not need to specify load addresses for each section but who 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 is of no particular importance. Remember, 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.

The **SORDER** command tells the linker to assign memory in the address order specified. The linker follows these rules for assigning memory space:

1. All absolute sections and those assigned addresses explicitly are placed in memory first.
2. The first section specified in the **SORDER** command is placed in memory at the first available location starting at the beginning of memory. The location is chosen with consideration for the alignment and size of the section.
3. All subsequent sections specified in the **SORDER** command(s) are placed in memory at the first available location (taking in consideration alignment and size of the section) after the previous **SORDER** section.
4. Sections named in the **SORDER** command with explicit addresses (rule #1) are checked to make sure that the specified order is maintained. The location of these sections will also set the starting point at which the linker

will begin looking for the starting point for next section named in the command.

5. If the order which the **SORDER** command specified cannot be maintained, the linker will issue a warning and will continue trying to assign memory to the remaining sections in the **SORDER** command.
6. If a section named in the **SORDER** command is not found, the section is ignored and the linker will continue with the next named section, if any.

The **ORDER** command applies to long sections; the **SORDER** command applies to short sections. If a section name appears in these commands for the first time it is assigned the appropriate length attribute, but it is assigned neither the common nor the noncommon attribute so that subsequently it can be either.

If the name of a long section appears in the **SORDER** command, a warning is printed and the section is given the short attribute. This can occur if a **SECT**, **COMMON**, **PAGE**, **CPAGE**, or **NOPAGE** command precedes the **SORDER** command, since these commands assign newly found sections the long attribute.

The same section name cannot appear twice in an **SORDER** command. Multiple **SORDER** commands are accepted without a warning. They are concatenated and must not contain duplicate names.

The order of sections within each group is specified by entering section names separated by commas. Any sections remaining within the group are assigned memory space after the sections specified in the command are ordered. The remaining sections will be placed in the order by which they are encountered by the linker.

An **SORDER** command can be continued to the next line by terminating it with one or more spaces followed by a pound sign (#). A continuation character must be placed like a comma between section names.

### Examples

```
ORDER SEC1, COMSEG
SORDER SEC2, SHORTSEC

SOORDER SECT1 #
SECT2, SECT3 #
SECT4
```

## START — Specifies Output Module Starting Address

### Syntax

```
START value | symbol
```

### Description

*value*              Specifies the starting address used in the output object module.

*symbol*              Specifies the starting address used in the output object module.

The **START** command specifies the absolute starting address to be placed in the terminator record of the object module. The starting address may be entered directly or symbolically.

If no starting address is specified, the starting address is obtained from the **END** record of the main program of the input modules, created using the **END** assembler directive. If no main program has been read, the starting address will be zero.

The **START** command can be useful if the starting address falls in an absolute section or in a relocatable section with a specified load address.

### Notes

When the load address is rounded upwards to lie on a word or page boundary, the starting address is not likewise rounded. If the symbol does not exist, an error is generated and the **START** command is ignored.

If a symbol is used to specify a starting address, it must be publicly defined. An error is generated if the symbol does not exist.

### Example

```
START $7FC
```

## SYMTRAN — Transforms Public/External Symbols

### Syntax

```
SYMTRAN /EMBEDDED "string1" [*string2*]
SYMTRAN [/LEADING] "string1" [*string2*]
SYMTRAN /TRUNCATE num
```

### Description

/EMBEDDED	Each occurrence of the character pattern " <i>string1</i> " (in the order it appears in the symbol) in each public symbol will have that pattern changed to " <i>string2</i> ". If " <i>string2</i> " is absent then all occurrences of the pattern in the symbol are simply deleted. This is useful when combining code written with different leading character conventions.
/LEADING	Each public symbol whose leading characters matches the pattern in " <i>string1</i> " will have that pattern changed to " <i>string2</i> ". If " <i>string2</i> " is absent then the leading pattern in the symbol is simply deleted. This is useful when combining code written with different leading character conventions.
/TRUNCATE	Truncate each public symbol whose length exceeds <i>num</i> .
<i>string1</i>	The character pattern to transform.
<i>string2</i>	The replacement character pattern.
<i>num</i>	The number at characters that symbols will be truncated to.

Since the mapping occurs at input, the original spelling of the public symbols is not retained. Only the altered public symbols appear in the map file or resulting object output file. Unique symbol names may not be unique after the SYMTRAN mapping.

Multiple SYMTRAN commands may be used. SYMTRAN commands apply during the same pass of the command file as LOAD and PUBLIC commands, and they take immediate effect. They apply only to the input characters mentioned in "*string1*", and do not alter any other mappings from prior SYMTRAN commands.

**Examples**

```
SYMTRAN /EMBEDDED          "_"
PUBLIC                   _SYM_1_      ; _SYM_1_ -> SYM1

SYMTRAN /EMBEDDED          "ABA" "X"
PUBLIC                   ABABABABABA ; ABABABABABA -> XBXBX

SYMTRAN /LEADING           "_"
PUBLIC                   _SYM_1_      ; _SYM_1_ -> SYM_1_

SYMTRAN
PUBLIC                   "_"_
._SYM_2_      ; SYM_2_ -> SYM_2

SYMTRAN
PUBLIC                   "ABA" "X"
ABABABABABA ; ABABABABABA -> XBABABABABA

SYMTRAN /TRUNCATE          4
PUBLIC                   _SYM_1_      ; _SYM_1_ -> _SYM
```

## UPPERCASE — Shifts Names to Uppercase

### Syntax

```
UPPERCASE [class [, class] ...]
```

### Description

<i>class</i>	Specifies one of the following class names:
PUBLICS	All public and external names.
MODULES	All module names.
SECTIONS	All section names.

The **UPPERCASE** command causes the linker to shift names to uppercase on input. All symbolic names of the specified classes will appear in uppercase in the linker's output files.

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**). **CASE**, **LOWERCASE**, and **UPPERCASE** take immediate effect and should be used 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 set to uppercase 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
...
```

## WARN — Modifies Message Severity

### Syntax

```
WARN number [, number] ...
```

### Description

*number*              Specifies a message number.

The **WARN** command specifies that the message indicated by *number* is to be treated as a warning.

This command has a global effect from the point at which the linker processes the information contained in the command. A subsequent **WARN** command overrides any values set by a previous one.

Fatal errors and the errors or warnings that are generated as a result of a syntactically or semantically incorrect **WARN** command cannot be overridden.

### Example

```
WARN 325
```

Downgrades error condition 325 to a warning.

# Sample Linker Session 11

---

## Introduction

The linker uses a two-pass process in which the commands and object modules are checked for errors. A symbol table is formed during pass one, and the output object module is generated on pass 2.

Errors detected during the first pass of processing will be displayed on the listing. If the linker is executed in batch mode, fatal errors cause the linker to terminate with the message **Load Not Completed**.

During pass two of processing, the final absolute object module is produced along with a link map and a listing of unresolved external references. A local symbol table, public symbol table, and cross-reference table map may optionally be listed on the link map. The link map also indicates the output module name and format, the section and module summary, and the starting address of the load. Detailed descriptions of the map file and listings can be found in this chapter.

## Linker Listings

Figure 11-1 shows the linker command file, and Figure 11-2 shows the output listing. Figure 11-3 shows the linker absolute file, while Figures 11-4 to 11-6 show the assembler files used. They are used as a reference to describe both the linker output listing format and the loading process.

The main purpose of the linker output is to convey all pertinent information regarding the linker process. The listing can also be used as a documentation tool by including comments and remarks that describe the function of the particular program section.

Refer to the following points in order to examine and understand the linker listing.

1. Figure 11-1 shows the command file which includes the following commands:
  - a. The **CHIP** command ensures that the linked code will run on a 68000 microprocessor.
  - b. The **LISTMAP** and **LISTABS** commands in this command file generate a symbol table listing of both local and external definition symbols and places the local symbols into the output object module. The symbol tables will display the symbols along with their final

absolute values. You can determine from the link map and symbol addresses as well as from the final object module that modules have been correctly linked together to form a final absolute module. All addresses will be adjusted to the correct values, and all links between modules resolved.

- c. The user has specified the starting address of the section named COMSEC.
  - d. The ORDER command places the sections in a different order than the default.
  - e. PUBLIC EXTRANEous=\$2000 defines the value of the EXTRANEous symbol to be \$2000.
  - f. NAME TESTCASE specifies the name of the final output object module as TESTCASE.
  - g. The link map will show the starting and ending addresses of the sections of the three modules in the order loaded. Note that a PAGE SECT2 command appears in the linker command file so the link map will show that the initial piece of SECT2 (from the first module) starts on the next page boundary.
  - h. The LOAD command will read the three modules from the files shown.
  - i. The END command starts the final steps in the load process.
2. Figure 11-2 shows the link map which shows the following:
- a. The commands in the linker command file.
  - b. The output module name and the output module format.
  - c. The names of all sections followed by the attribute, starting address, ending address, length, and type of alignment for each section.
  - d. 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 two of the load are indicated at the end of the module summary.
3. The unresolved externals section contains a list of the undefined external references.

4. When the appropriate **LISTMAP** 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. Symbols are displayed as follows:

Public symbols as declared in the assembler are external definition symbols and are used for intermodule communication.

Local symbols are those known only to a single module. Local symbols are not used by the linker, but are listed so their final absolute values can be seen. The attributes and sections are listed for each local symbol, as well as the section offsets and modules which define them. Since **LISTMAP CROSSREF** is specified, a cross-reference table is listed. Local symbols can be placed in the output object module of the assembler by specifying the **OPT T** directive, and can subsequently be used for symbolic debugging.

5. 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.
6. The public symbol table contains the list of **PUBLIC** symbols, the section, the actual section address, and the modules.
7. The cross-reference option is turned off by default. To produce the cross-reference table, use the **LISTMAP CROSSREF** command. An example of the cross-reference table output is shown in the map file after the symbol table information. All external symbols passed to the linker are listed under the heading **SYMBOL**. The symbol section and address are listed. Any flag to the left of those values is the section 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. In this example, Load Completed is shown.

## Sample Linker Listing

### Linker Command File

1 →

```
CHIP 68000
LISTMAP INTERNALS,PUBLICS,CROSSREF
LISTABS INTERNALS,PUBLICS
COMMON COMSEC=$1000
ORDER SECT2,SECT3,COMSEC
PUBLIC EXTRANEous=$2000
NAME TESTCASE
PAGE SECT2
* Load first two modules
LOAD lnk68ka.obj,lnk68kb.obj
* Load last module
LOAD lnk68kc.obj
END
```

Figure 11-1. Linker Command File

## Linker Map File

```
Microtec Research LNK68K Version x.y   Thu Jul 23 09:19:05 1992   Page  1
Command line: /usr4/engr.sun4/bin/lnk68k -c tst68k.opt -Fs -o cmp68k.x -m

2 → CHIP 68000
LISTMAP INTERNALS,PUBLICS,CROSSREF
LISTABS INTERNALS,PUBLICS
COMMON COMSEC=$1000
ORDER SECT2,SECT3,COMSEC
PUBLIC EXTRANEOUS=$2000
NAME TESTCASE
PAGE SECT2
* Load first two modules
LOAD lnk68ka.obj,lnk68kb.obj
* Load last module
LOAD lnk68kc.obj
END
```

Figure 11-2. Linker Map File

2 →

```

Microtec Research LNK68K Version x.y      Thu Jul 23 09:19:06 1992      Page 2

OUTPUT MODULE NAME: TESTCASE
OUTPUT MODULE FORMAT: MOTOROLA S2

SECTION SUMMARY
-----
SECTION ATTRIBUTE          START      END        LENGTH      ALIGN
SECT2  NORMAL CODE         00000000  00000225  00000226  2 (WORD)
SECT3  NORMAL CODE         00000226  0000023F  0000001A  2 (WORD)
SECT1  SHORT DATA          00000240  00000292  00000053  2 (WORD)
          ABSOLUTE DATA       00000400  00000405  00000006  0 (BYTE)
          ABSOLUTE CODE        00001000  00001009  00000004  0 (BYTE)
COMSEC  COMMON              00001000  00001001  00000002  2 (WORD)

MODULE SUMMARY
-----
MODULE      SECTION:START      SECTION:END      FILE
MAIN1      SECT1:00000240      SECT1:00000290  /test/lnk68ka.obj
          :00001000            :00001009
          SECT2:00000000        SECT2:00000059
READ       SECT1:00000290      SECT1:00000292  /test/lnk68kb.obj
          SECT2:00000100        SECT2:00000189
          COMSEC:00001000       COMSEC:00001000
VERYVERYLO SECT2:00000200      SECT2:00000225  /test/lnk68kc.obj
          SECT3:00000226        SECT3:0000023F
          :00000400            :00000405
          COMSEC:00001000       COMSEC:00001001

```

3 →

```

ERROR: (320) UNRESOLVED EXTERNALS:
-----
```

SYMBOL	MODULE
SCAN	MAIN1

4, 5, 6 →

```

LOCAL SYMBOL TABLE
-----
-----
```

SYMBOL	ATTRIB	SECTION	OFFS/ADDR	MODULE:FUNCTION
UDATOUT	ASMVAR	ABSCONST	00F7001F	MAIN1:
USTAT	ASMVAR	ABSCONST	00F70031	MAIN1:
MAIN10	ASMVAR	SECT2	0000000C	MAIN1:
UDATIN	ASMVAR	ABSCONST	00F70019	MAIN1:

Figure 11-2. Linker Map File (cont.)

4, 5, 6 →

Microtec Research LNK68K Version x.y			Thu Jul 23 09:19:06 1992	Page 3
ASLF	ASMVAR	ABSCONST	0000000A	MAIN1:
DUT8	ASMVAR	SECT2	00000034	MAIN1:
IN8	ASMVAR	SECT2	0000001C	MAIN1:
PROC	ASMVAR		00001004	MAIN1:
MAIN2	ASMVAR	SECT2	00000000	MAIN1:
BLNK	ASMVAR	ABSCONST	00000014	MAIN1:
ASCR	ASMVAR	ABSCONST	0000000D	MAIN1:
LENGTH	ASMVAR	COMSEC	00001000	READ:
READ40	ASMVAR	SECT2	00000146	READ:
BSPA	ASMVAR	ABSCONST	00000008	READ:
COUNT	ASMVAR	SECT1	00000292	READ:
READ50	ASMVAR	SECT2	0000015C	READ:
READ60	ASMVAR	SECT2	0000016C	READ:
READ10	ASMVAR	SECT2	00000184	READ:
READ70	ASMVAR	SECT2	00000172	READ:
TAB	ASMVAR	ABSCONST	00000009	READ:
READ20	ASMVAR	SECT2	00000120	READ:
READ80	ASMVAR	SECT2	0000017C	READ:
BLNK	ASMVAR	ABSCONST	00000020	READ:
ASCR	ASMVAR	ABSCONST	0000000D	READ:
READ30	ASMVAR	SECT2	00000138	READ:

PUBLIC SYMBOL TABLE

SYMBOL	SECTION	ADDRESS	MODULE
COMSEC	COMSEC	00001000	\$\$
CRLF	SECT2	00000048	MAIN1
DLETE		0000007F	\$\$
ECHO	SECT1	00000290	MAIN1
EXTRANEOUS		00002000	\$\$
INBUF	SECT1	00000240	MAIN1
INBUFEND	SECT1	00000290	MAIN1
READ	SECT2	00000100	READ
TIN	SECT2	0000001C	MAIN1
TOUT	SECT2	00000034	MAIN1

CROSS REFERENCE TABLE

7 →

SYMBOL	SECTION	ADDRESS	MODULE
COMSEC	COMSEC	00001000	-\$\$
CRLF	SECT2	00000048	-MAIN1
DLETE		0000007F	READ
ECHO	SECT1	00000290	-MAIN1
EXTRANEOUS		00002000	READ
INBUF	SECT1	00000240	-MAIN1

Figure 11-2. Linker Map File (cont.)

8, 9 →

Microtec Research LNK68K Version x.y   Thu Jul 23 09:19:06 1992   Page 4			
INBUFEND	SECT1	00000298	READ
READ	SECT2	00000100	-MAIN1 -READ
SCAN		00000000	MAIN1 -\$
TIN	SECT2	0000001C	-MAIN1 READ
TOUT	SECT2	00000034	-MAIN1 READ

START ADDRESS: 00000000  
Load Completed  
  
Errors: 1, Warnings: 0

Figure 11-2. Linker Map File (cont.)

## **Linker Absolute File**

The absolute output module is shown in this section. In this sample, the output object file shown is in Motorola S-record absolute hexadecimal format.

**Figure 11-3. Linker Absolute File**

## Assembly Listings for the Linker Example

Figures 11-4 to 11-6 show three assembly listings of programs that are combined by the linker. The actual load is shown in Figure 11-4. Note the following points when examining the assembly listings:

1. The **MAIN** program contains references to subroutines called **READ** and **SCAN** that are not in the program but are declared external. **SCAN** is not defined in any module. The user could have specified the address of **SCAN** at load time with a **PUBLIC** command.
2. The second assembly listing shows the **READ** module that is required by the **MAIN** program and also shows that **READ** references the I/O drivers **TIN** and **TOUT** that are declared external in the **MAIN** program.
3. The third program contains no links to the other programs, but does contain absolute code. The **common** section is used to provide linkage between the programs.

## Assembly Listing 1

```

Microtec Research ASM68K Version x.y      Wed Jul 22 17:08:16 1992    Page 1
Command line: /usr4/engr.sun4/bin/asm68k -l lnk68ka.tst
Line Address
1          *
2          * This is a sample program that demonstrates some of the
3          * features of the relocatable assembler and linker.
4          * Three modules are linked together to form the final
5          * program. External definitions and references as well
6          * as COMMON are used to communicate between routines.
7          *
8          * The main program is below. In the main program are
9          * I/O drivers that are referenced by one of the routines
10         * linked to this main program.
11         *
12             LLLEN    100
13             MAIN1   IDNT
14                 OPT    D,E,T
15         *
16         *
17             XDEF    INBUF,INBUFEND,TIN,TOUT,CRLF,ECHO,DLETE
18             XREF    READ,SCAN
19         *
20             SECT.S SECT1
21 00000000     INBUF   DS.B  80           ;Input buffer
22             INBUFEND
23 00000050     ECHO    DS.B  1            ;Echo flag
24         *
25             ORG    $1000
26 00001000 6000 0002     BRA    PROC
27 00001004 2E7C 0000 1000     PROC   MOVE.L #\$1000,SP    ;Set stack
28         *
29             SECT   SECT2
30 00000000 4EB9 0000 0000     E     MAIN2  JSR    READ
31 00000006 227C 0000 0000     R     MOVE.L #INBUF,A1    ;Read next line
32 0000000C 1219             MAIN10 MOVE.B (A1)+,D1    ;Start of buffer
33 0000000E 0C01 0014             CMP.B #BLNK,D1    ;Check for non-blank
34 00000012 67F8             BEQ    MAIN10
35 00000014 4EB9 0000 0000     E     JSR    SCAN
36 0000001A 4E75             RTS
37         *
38         * NAME   - INB
39         *
40         * THIS ROUTINE WILL READ A CHARACTER FROM THE TERMINAL
41         *
42         * ENTRY PARAMETERS
43         *
44         * NONE
45         *
46         * EXIT PARAMETERS
47         *
48         * D2 - INPUT CHARACTER
49         *
50         *

```

Figure 11-4. Assembly Listing 1

```

Microtec Research ASM68K Version x.y      Wed Jul 22 17:00:15 1992    Page 2

Line Address
51 0000001C 3239 00F7 0031    IN8   MOVE.W USTAT,D1          ;Read UART status
52 00000022 0001 0001          BTST.L #$1,D1          ;Check if ready
53 00000026 66F4              BNE   INB           ;Not ready yet
54 00000028 1439 00F7 0019    MOVE.B UDATIN,D2        ;Read data
55 0000002E 0202 007F          ANDI.B #DELETE,D2      ;Delete parity bit
56 00000032 4E75              RTS

57 *
58 * NAME - OUT8
59 *
60 * THIS ROUTINE WRITES A CHARACTER TO THE TERMINAL
61 *
62 * ENTRY PARAMETERS
63 *
64 * D2 - CHARACTER TO OUTPUT
65 *
66 * EXIT PARAMETERS
67 *
68 * NONE
69 *

70 00000034 3239 00F7 0031    OUT8  MOVE.W USTAT,D1          ;Read status
71 0000003A 0001 0002          BTST.L #2,D1          ;Check if ready
72 0000003E 66F4              BNE   OUT8          ;Not ready
73 00000048 13C2 00F7 001F    MOVE.B D2,UDATOUT       ;Output data
74 00000046 4E75              RTS

75 *
76 * NAME - CRLF
77 *
78 * THIS ROUTINE OUTPUTS A CARRIAGE RETURN AND LINE FEED
79 *
80 *

81 00000048 143C 0000    CRLF   MOVE.B #ASCR,D2
82 0000004C 4EBA FF65          JSR    OUT8
83 00000050 143C 000A          MOVE.B #ASLF,D2
84 00000054 4EBA FFDE          JSR    OUT8
85 00000058 4E75              RTS

86 *
87 *
88 00F70031    USTAT   EQU    $F70031          ;UART status
89 00F7001F    UDATOUT EQU    $F7001F          ;Serial output port
90 00F70019    UDATIN  EQU    $F70019          ;Serial input port
91 0000000D    ASCR    EQU    13
92 00000024    ASLF    EQU    10
93 00000014    BLNK    EQU    20
94 0000007F    DELETE  EQU    $7f
95 0000001C    RTIN    EQU    IN8
96 00000034    RTOUT   EQU    OUT8
97                                     END     MAIN2

```

Figure 11-4. Assembly Listing 1 (cont.)

Symbol Table	
Label	Value
ASCR	00000000
ASLF	0000000A
BLNK	00000014
CRLF	SECT2:00000048
DLETE	0000007F
ECHO	SECT1:00000050
IN8	SECT2:0000001C
INBUF	SECT1:00000000
INBUFEND	SECT1:00000050
MAIN10	SECT2:0000000C
MAIN2	SECT2:00000000
OUT8	SECT2:00000034
PROC	00001004
READ	External
SCAN	External
TIN	SECT2:0000001C
TOUT	SECT2:00000034
UDATIN	00F70019
UDATOUT	00F7001F
USTAT	00F70031

Figure 11-4. Assembly Listing 1 (cont.)

## Assembly Listing 2

2 →

```

Microtec Research ASM68K Version x.y      Thu Jul 23 09:49:04 1992 Page 1

Command line: /usrd4/engr.sun4/bin/asm68k -l lnk68kb.tst
Line Address
1          *
2          * NAME - READ
3          *
4          * THIS ROUTINE READS IN A LINE FROM THE TERMINAL AND
5          * PLACES IT INTO THE INPUT BUFFER. THE FOLLOWING
6          * SPECIAL CHARACTERS ARE RECOGNIZED:
7          *
8          *   CR           - END OF LINE
9          *   CONTROL X    - DELETE CURRENT LINE
10         *   DEL          - DELETE LAST CHARACTER
11         *
12         *   ALL DISPLAYABLE CHARACTERS BETWEEN BLANK AND DEL
13         * EXCEPT FOR THE ABOVE SPECIAL CHARACTERS ARE
14         * RECOGNIZED BY THIS * ROUTINE, AS WELL AS THE TAB.
15         * ALL OTHER CHARACTERS ARE IGNORED. AN ATTEMPT TO
16         * READ MORE CHARACTERS THAN ALLOWED IN THE INPUT
17         * BUFFER WILL BE INDICATED BY A BACKSPACE.
18         *
19         * ENTRY PARAMETERS
20         *
21         * ECHO - ECHO FLAG, B= NO ECHO
22         *
23         * EXIT PARAMETERS
24         *
25         * INBUF - CONTAINS INPUT LINE
26         *
27         *
28         LLEN     80
29
30         READ    IDNT
31             OPT      D,E,T
32             XDEF    READ
33             XREF   CRLF,TIN,TOUT
34             XREF.S  INBUF,INBUFEND,ECHO,DELETE
35         *
36         *
37     00000000  SECT.S  SECT1
38         COUNT   DS.B    1
39         *
40         *
41         SECT    SECT2
42     00000000 227C 0000 0000  E READ   MOVE.L  #INBUF,A1
43     00000006 4278 0000        R CLR     COUNT
44     0000000A 4EB9 0000 0000  E READ10 JSR     TIN
45     00000010 0C41 0018        CMP     #24,D1
46     00000014 6500 000A        BNE     READ20
47     00000018 4EB9 0000 0000  E JSR     CRLF
48     0000001E 50E8        BRA     READ
49     00000020 0C41 000D        READ20 CMP     #ASCR,D1
50     00000024 6500 0012        BNE     READ30

```

Figure 11-5. Assembly Listing 2

```

Microtec Research ASM68K Version x.y    Thu Jul 23 09:49:04 1992    Page 2

Line Address
51 00000028 3438 0000      R      MOVE.W   COUNT,D2
52 0000002C 67DC          BEQ    READ10
53 0000002E 2281          MOVE.L   D1,(A1)
54 00000030 33C2 0000 0000  R      MOVE.W   D2,LENGTH
55 00000036 4E75          RTS
56 00000038 0C01 0000      E      READ30  CMP.B    #DELETE,D1
57 0000003C 6600 001E          BNE    READ050
58 00000040 3438 0000      R      MOVE.W   COUNT,D2
59 00000044 67C4          BEQ    READ10
60 00000046 5389          READ40  SUB.L    #1,A1
61 00000048 0438 0001 0000  R      SUBI.B  #1,COUNT
62 0000004E 343C 0000          MOVE    #BSPA,D2
63 00000052 4EB9 0000 0000  E      JSR     TOUT
64 00000058 6000 0018          BRA    READ070
65 0000005C 0C01 0009      READ50  CMP.B    #TAB,D1
66 00000060 5700 000A          BEQ    READ060
67 00000064 0C01 0020          CMP.B    #BLNK,D1
68 00000068 6000 0008          BLT    READ070
69 0000006C 12C1          READ60  MOVE.B   D1,(A1)+_
70 0000006E 5278 0000      R      ADD     #1,COUNT
71 00000072 3438 0000      R      READ70  MOVE    COUNT,D2
72 00000076 0C42 0050          CMPI   #80,D2
73 0000007A 67CA          BEQ    READ40
74 0000007C 3238 0000      E      READ60  MOVE.W   ECHO,D1
75 00000080 5700          BEQ    READ10
76 00000082 4EB9 0000 0000  E      JSR     TOUT
77 00000088 6000          BRA    READ10
78                                * PLACE VARIABLES IN COMMON
79                                COMMON  COMSEC
80 00000000          LENGTH DS.B    1
81 00000000          ASCR   EQU     13
82 00000008          BSPA   EQU     8
83 00000028          BLNK   EQU     $20
84 00000009          TAB    EQU     $09
85                                END

```

Figure 11-5. Assembly Listing 2 (cont.)

Symbol Table	
Label	Value
ASCR	00000000
BLNK	00000020
BSPA	00000008
COUNT	SECT1 :00000000
CRLF	External
DELETE	External
ECHO	External
INBUF	External
INBUFEND	External
LENGTH	COMSEC:00000000
READ	SECT2 :00000000
READ10	SECT2 :0000000A
READ20	SECT2 :00000020
READ30	SECT2 :00000038
READ40	SECT2 :00000046
READ50	SECT2 :0000005C
READ60	SECT2 :0000006C
READ70	SECT2 :00000072
READ80	SECT2 :0000007C
TAB	00000009
TIN	External
TOUT	External

Figure 11-5. Assembly Listing 2 (cont.)

### Assembly Listing 3

```

Microtec Research ASM68K Version x.y    Thu Jul 23 09:37:30 1992 Page 1
Command line: /usr4/engr.sun4/bin/asm68k -l lnk68kc.tst
Line Address
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49

      * NAME - VERYVERYLONGMODULENAME
      *
      * THIS ROUTINE DOES NOT HAVE ANY EXTERNAL
      * REFERENCES TO THE OTHER MODULES, BUT
      * CREATES SEVERAL RELOCATABLE SEGMENTS
      * AND IS LINKED WITH THEM FOR DEMONSTRATION
      * PURPOSES.
      *
      * VERYVERYLONGMODULENAME IDNT
      OPT E
      *
      * Create a Relocatable Section
      *
      SECT   SECT2
      NOP
      MOVE.W LEN1,D1
      ADD DATA,D1
      MOVE D1,LEN2
      BPL LAB1
      EORI.B #$AA,D1
      MOVE.B DATA+S,D1
      TRAP #15
      BNE LAB3
      *
      * Create a Relocatable Section
      *
      SECT   SECTS
      MOVE.L #DATA,A1
      MOVE (A1)+,D1
      BNE SKP
      JSR LAB1
      DC.L LAB1
      DC.L DATA
      *
      * Create an Absolute Section
      *
      ORG $400
      DATA DC.B 5,6
      R DC LAB1
      NOP
      *
      * Place variables in COMMON
      *
      COMMON COMSEC
      LEN1 DS.B 1
      LEN2 DS.B 1
      END
  
```

3 →

Figure 11-6. Assembly Listing 3

Symbol Table	
Label	Value
DATA	00000400
LAB1	SECT2 :0000001C
LAB3	SECT2 :00000022
LEN1	COMSEC:00000000
LEN2	COMSEC:00000001
SKP	SECT3 :00000012

Figure 11-6. Assembly Listing 3 (cont.)

# Librarian Operation 12

---

## Introduction

The LIB68K Object Module Librarian builds program libraries. Libraries are collections of relocatable object modules residing in a single file. These libraries cause the linker to automatically load frequently used object modules that define public symbols referenced in other loaded modules. These modules are linked without concern for the specific names and characteristics of the modules in which the symbols are defined.

The LIB68K Object Module Librarian features include:

- Creation of libraries that can be loaded by the linker
- Ability to add, delete, or replace individual modules in a library
- Ability to display library directories
- Case-sensitivity for symbol names
- Batch command-line input and return codes for make-type utilities
- Interactive operation
- Optimized library structure for fast access during a link

This chapter describes how to build and modify the libraries and how the linker uses the libraries.

When used in connection to the librarian, the word "module" refers to a relocatable object module that results from assembling a source program with the ASM68K Assembler.

Error messages and warnings are listed in Appendix D, *Librarian Error Messages*.

## Librarian Function

The librarian formats and organizes library files used by the linker. Libraries provide a convenient means for managing collections of relocatable object modules. Through the use of libraries, linkers can easily access relocatable object modules when required. This efficiency comes from reducing the number of files that must be opened for linking modules.

When writing modular programs, communication among the various modules is established through the use of public and external symbols. For example, Figure 12-1 shows three relocatable object modules that result from an assembly.

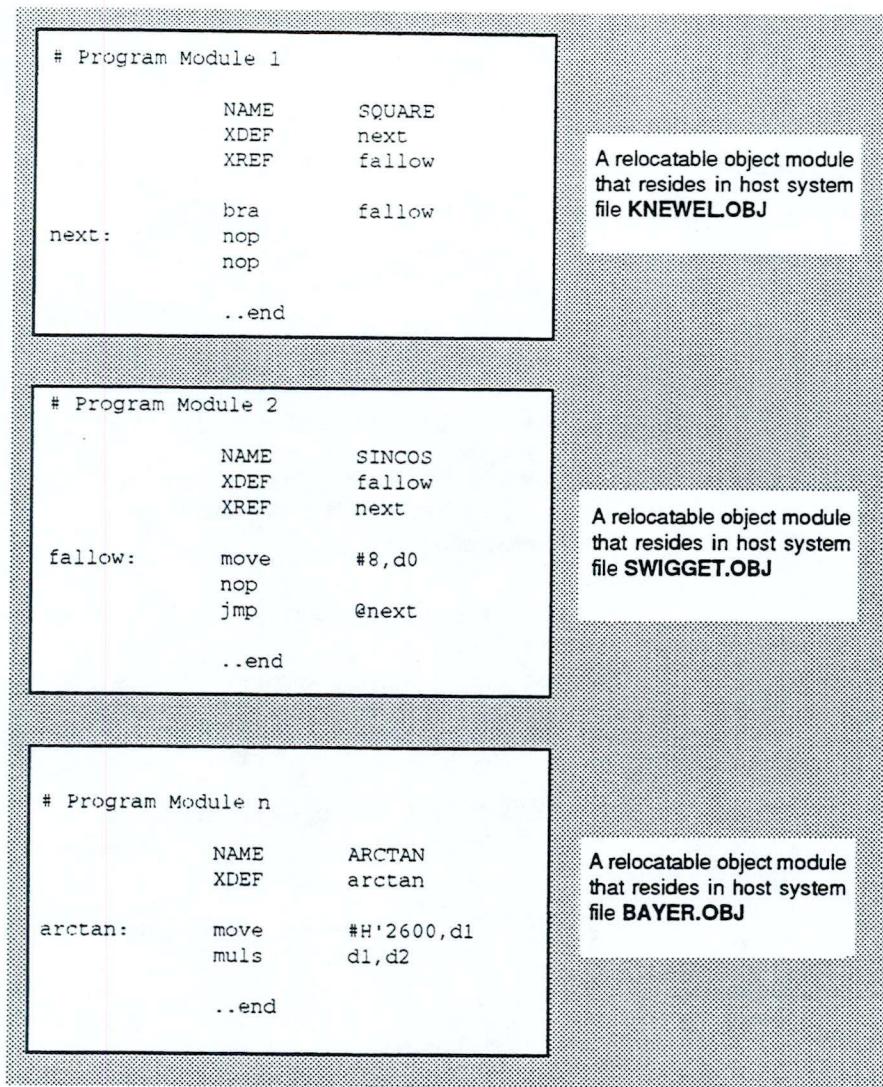


Figure 12-1. Three Relocatable Object Modules Resulting from Assembly

Of the three modules shown, Program Modules 1 and 2 communicate with one another through external references and public symbols, while Program Module 3 is a stand-alone module.

The relocatable modules illustrated consist of load data information, relocation

information, and records that indicate public symbols and external symbols.

By using various combinations of librarian commands, the relocatable object modules shown can be made members of a library. There are three ways to invoke the LIB68K Object Module Librarian:

1. Using the command line.

Only certain library functions can be specified using this method:  
**ADDMOD, DELETE, REPLACE, EXTRACT, and FULLDIR.** These librarian commands can be entered in any order, but the librarian processes the commands in the order specified above.

2. Reading librarian commands from a command file in batch mode.

The commands are read in the exact order in which they are specified. If an error is found, commands read after the first error is encountered are processed, checked for errors, and executed if possible. However, a library file, if specified, is not generated.

3. Entering librarian commands interactively from the terminal.

When an illegal command is entered, the librarian displays an error message and provides an opportunity to reenter the command. In this interactive mode, most librarian command errors are not fatal.

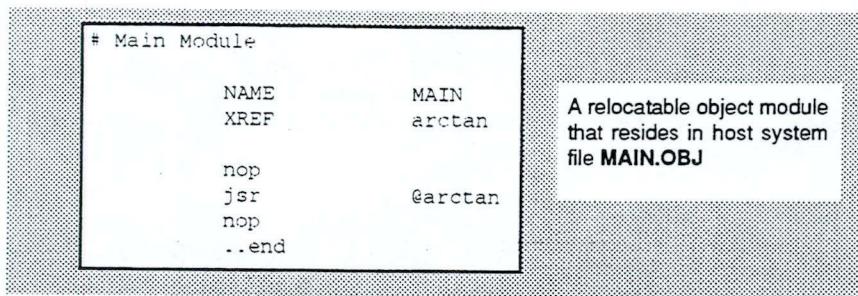
The relocatable modules illustrated consist of load data information, relocation information, and records that indicate public symbols and external symbols.

For example, a new library can be created by using the following commands (the *User's Guide* contains a full description of librarian invocation):

```
CREATE  NEWREM.LIB
ADDMOD  KNEWEL.OBJ
ADDMOD  SWIGGET.OBJ, BAYER.OBJ
SAVE
```

Now the library can be used by the linker.

Assume that you have written a program module called **MAIN**. After **MAIN** has been assembled, the resultant relocatable object module in Figure 12-2 is in a host system file named **MAIN.OBJ**. This module has a reference to the public symbol **ARCTAN**.

**Figure 12-2. Relocatable Object Module in MAIN.OBJ**

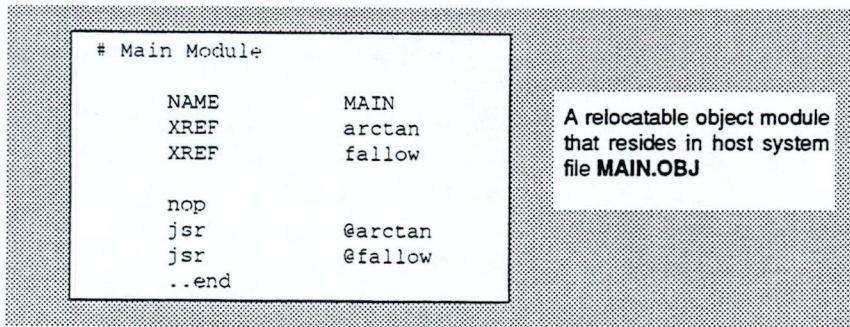
Before the library existed, you could have directed the linker to load the MAIN module and the module that contains the reference to ARCTAN as follows:

```
LOAD  MAIN.OBJ
LOAD  BAYER.OBJ
```

After the library has been created, you can direct the linker as follows:

```
LOAD  MAIN.OBJ
LOAD  NEWREM.LIB
```

The linker will access the library to try to resolve external references such as ARCTAN. The MAIN module can be modified so it calls the SINCOS module as well (see Figure 12-3).

**Figure 12-3. MAIN Module Modified to Call SINCOS Module**

Without the ability to link from a library, it would be necessary to command the linker as follows:

```
LOAD  MAIN.OBJ  
LOAD  SWIGGET.OBJ  
LOAD  BAYER.OBJ
```

However, when using a linker that has the ability to load from a library, you need specify only:

```
LOAD  MAIN  
LOAD  NEWREM.LIB
```

The linker will load the relocatable object module **MAIN** in the usual way. It will load the other modules from the library referenced by **MAIN**.

The following example is a more practical illustration of using the library.

**Example:**

Suppose you write a series of program modules consisting of a number of mathematical routines including a few modules that calculate transcendental functions. These modules are then gathered into a library file by the LIB68K Object Module Librarian.

Sometime later, you need to calculate an arc tangent function within a program being written. You are aware of the fact that there is an arc tangent function in a library file, and you know the name of the entry point of the routine. You also know how to pass parameters to the arc tangent function and how to accept the result of the calculation.

You need only do the following:

1. Call the arc tangent function from the program being developed, placing the public name of the entry point into the argument field of the **JSR** or **BRA** instruction.
2. Place the public entry point name of the arc tangent function in the argument field of an external reference (**XREF**) pseudo-op in the program being written.

Even though you do not know the name of the relocatable object module that contains the arc tangent function, you can include the correct relocatable object module by informing the linker to use the required library file.

You do not have to specify which module contains the arc tangent function. The linker automatically searches the named library. It looks for the entry point name coded as the argument of the calling statement. When the entry point name has been

found, the linker identifies the module in which it resides and then includes the module containing the name in the current load.

The linker determines which of the library modules to use by examining the internal list of unresolved external references accumulated during the link process. It then accesses the library file to determine if there is a match between unresolved external references, and a label or name that has been declared public in the library file modules. The linker then identifies which modules contain the matching public symbols and includes those modules just as if you had explicitly directed the linker to load the proper modules.

When the inclusion of a module in the library adds an undefined reference to the list of undefined references, the linker will access the library again until all external references have been satisfied. All public symbols within a library must have unique names.

## Return Codes

The librarian provides operating-system specific return codes. The librarian either completes without encountering an error, displays a message or warning, or terminates with an error. Error messages and warnings are listed in Appendix D, *Librarian Error Messages*. Return codes are described in the *Microtec Research ASM68K User's Guide*.

# Librarian Commands 13

## Introduction

This chapter describes the commands used by the LIB68K Object Module Librarian. The librarian reads a sequence of commands from the command input device in interactive or batch mode. The command sequence must be terminated by the **END** command. Relocatable object modules are read as input and collected in organized libraries as specified in the command input file.

Multiple sessions are permitted through the use of commands. A session is a list of commands before a **CLEAR** command. It does not end until this command is given. The **END** command terminates the current session and exits the librarian.

## Command Syntax

Librarian module names are written according to the rules for assembler module names. Each module must have a unique name. Public symbols are written according to the same definition as in the assembler.

The librarian recognizes the special characters listed in Table 13-1.

Table 13-1. Special Characters Recognized by the Librarian

*	asterisk	+	plus
,	comma	)	right parenthesis
(	left parenthesis	;	semicolon

The special characters perform the following functions when used in a library command line:

- The asterisk (\*) or semicolon (;) places a comment in a command sequence. The librarian ignores the rest of the line following these special characters. The librarian does not process comments; it writes them to the output file.
- The comma (,) separates members of a list of similar elements. The list can contain module names or module file names.
- The left and right parentheses ( ), used in pairs, denote a list of similar elements in a command. Parentheses can be used to group module names that are members of a library only.

- The plus sign (+) followed by a carriage return acts as a continuation character by allowing you to continue a list on one or more subsequent lines. Care should be exercised when using line continuation: do not break up or interrupt a complete syntactical unit such as a file name, a module name, or a command. The command verb must be terminated by a blank if it is 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 can appear anywhere in a command.

## Blanks

Except as noted above, blanks can be used freely within commands (between syntactically identifiable units).

### Example:

```
DELETE MOD1 , MOD2
```

is the same as:

```
DELETE MOD1,MOD2
```

## Command File Comments

Comments can 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.obj ; this is a command line comment
addmod moduleb      * this is another comment
```

## Command Summary

The following pages describe the librarian commands. An alphabetical listing of the librarian commands is shown in Table 13-2.

Table 13-2. Librarian Commands and Abbreviations

Abbreviation	Command	Description
ADDL	ADDLIB	Adds Module(s) from Another Library
ADDM	ADDMOD	Adds Object Module(s) to Current Library
CL	CLEAR	Clears Library Session Since Last SAVE
CR	CREATE	Creates New Library
DE	DELETE	Deletes Module(s) From Current Library
DI	DIRECTORY	Lists Library Modules
E / Q	END / QUIT	Terminates Librarian Execution
EXT	EXTRACT	Copies Library Module to a File
F	FULLDIR	Displays Library or Library Module Contents
H	HELP	Displays Context-Sensitive Command Syntax
OP	OPEN	Opens an Existing Library
R	REPLACE	Replaces Library Module
S	SAVE	Saves Contents of Current Library

## ADDLIB — Adds Module(s) from Another Library

### Abbreviation

ADDL

### Syntax

```
ADDLIB library_filename [ (module_name [, module_name] . . . ) ]
```

### Description

*library\_filename*      Specifies the library where the named modules reside.

*module\_name*      Indicates the relocatable object module(s) to be added to the library named in a previous OPEN or CREATE command.

You can include the entire library by entering the *library\_filename* and no module names.

The ADDLIB command adds one or more object modules from one library to the library currently being created or modified.

The OPEN or CREATE command must precede the ADDLIB command and name the library to which the object modules will be added.

### Related Commands

CREATE, OPEN

### Example

```
OPEN LIBRARY1.LIB
ADDLIB MATH.LIB(SQUARE, SQROOT)
SAVE
...
```

In the example above, LIBRARY1.LIB is opened. The ADDLIB command adds the modules named SQUARE and SQROOT from the library MATH.LIB to LIBRARY1.LIB. No changes occur to MATH.LIB. If SAVE is not entered, no changes will occur to LIBRARY1.LIB.

## ADDMOD — Adds Object Module(s) to Current Library

### Abbreviation

ADDM

### Syntax

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

### Description

*filename*      Specifies the file to be added to the library named in the **OPEN** or **CREATE** command.

The **ADDMOD** command adds a nonlibrary file containing one or more relocatable object modules to the library named in the **OPEN** or **CREATE** command.

The module(s) added to the library should have been named at assembly time with the **NAME** directive. The **OPEN** or **CREATE** command must precede the **ADDMOD** command.

### Related Commands

**CREATE**, **OPEN**

### Example

```
OPEN LIBRARY2.LIB
ADDMOD MATH.MBR
SAVE
...
```

In the example above, the **ADDMOD** command adds the file **MATH.MBR** to the library named in the **OPEN** command, **LIBRARY2.LIB**.

## CLEAR — Clears Library Session Since Last SAVE

### Abbreviation

CL

### Syntax

CLEAR

### Description

The CLEAR command clears all commands that have been entered in the current library session since the last SAVE command.

### Related Commands

SAVE

### Example

```
OPEN    LIBRARY2.LIB
ADDMOD MATH.O
SAVE
OPEN    WRONG_LIB.LIB
ADDMOD FONT.O
CLEAR
OPEN    LIBRARY3.LIB
...
```

In the example above, CLEAR must be executed before opening and processing LIBRARY3.LIB.

## CREATE — Creates New Library

### Abbreviation

CR

### Syntax

CREATE *library\_name*

### Description

*library\_name*      Specifies the name of the library file being created.

The CREATE command creates a new library. Naming conventions should follow those of your host computer and operating system. You can create only one library at a time. A newly created library must be saved before a second one is created.

It is an error to add, replace, delete, or extract modules from a nonexistent library. It is also an error to create an existing library. If *library\_name* exists in the current directory, a warning message will be issued in interactive mode; in batch mode, an error will be issued and the new library will not be saved.

### Related Commands

OPEN

### Example

```
CREATE TEMPOR.LIB
```

```
...
```

In the example above, the command CREATE TEMPOR.LIB creates a library file called TEMPOR.LIB.

If TEMPOR.LIB already exists, a warning is displayed in interactive mode. When using the librarian in batch or command line mode, and you name an existing library with the CREATE command, the librarian issues an error message. No library is created.

## DELETE — Deletes Module(s) from Current Library

### Abbreviation

DE

### Syntax

DELETE *module\_name*[,*module\_name*]...

### Description

*module\_name*      Specifies the name of the module to be removed from the library named in a previous OPEN or CREATE command.

The DELETE command removes one or more relocatable object modules from the library named in the OPEN or CREATE command. Object module names are case-sensitive.

An OPEN or CREATE command must precede DELETE.

### Related Commands

CREATE, OPEN

### Example

```
OPEN LIBRARY3.LIB
DELETE ARCTAN, SQUARE, RAD
SAVE
...
```

In the example above, the DELETE command deletes relocatable object modules named ARCTAN, SQUARE, and RAD from the library named LIBRARY3.LIB.

## DIRECTORY — Lists Library Modules

### Abbreviation

DI

### Syntax

```
DIRECTORY library_name [ (module_name [, module_name] . . . ) ] [list_filename]
```

### Description

*library\_name*      Specifies the name of the library whose module names and sizes are to be listed.

If you enter just the *library\_name*, all modules are listed.

*module\_name*      Specifies the name of the module whose size is to be listed.

When you enter specific *module\_name*, directory information is displayed for the named modules only.

*list\_filename*      Writes the directory information to the named file. If not specified, the output defaults to the standard list device, usually the terminal.

The **DIRECTORY** command lists module names and sizes of the modules in the specified library. The sizes listed are the number of bytes required to store the modules on the host computer system.

Object module names are case-sensitive.

### Related Commands

FULLDIR

### Example

```
DIRECTORY sieve.lib
```

In the example above, all modules in *sieve.lib* and their sizes are listed:

```
Library sieve.lib
Module      Size Processor
SIEVE ..... 386   68000
MODULE1 ... 397   68000
MODULE .... 289   68000

Module Count = 3
```

## END / QUIT — Terminates Librarian Execution

### Abbreviation

EN  
Q

### Syntax

END  
QUIT

### Description

The END or QUIT command terminates librarian command processing without building a new library.

If you do not issue a SAVE command before END or QUIT, the librarian will not build a new library.

### Related Commands

SAVE

### Example

```
DIR NEW.LIB
END
```

In the example above, the librarian lists the contents of the library NEW.LIB. The END command exits the library. Since there is no SAVE command, a new library is not built.

## EXTRACT — Copies Library Module to a File

### Abbreviation

EXT

### Syntax

EXTRACT *module\_name* [, *module\_name*] . . .

### Description

*module\_name*      Specifies the module to be copied from the library named in a previous OPEN or CREATE command.

The EXTRACT command copies a library module to a file outside the library. The extracted module contains the host-specific path and file name specification in the same format as that generated by the assembler. Consequently, it can be explicitly loaded.

An OPEN or CREATE command must precede the EXTRACT command.

### Related Commands

CREATE, OPEN

### Example

```
OPEN    LIBASC.LIB
EXTRACT MODA,MODB,MODC
...
```

In the example above, the modules MODA, MODB, and MODC are extracted from the current library and written to files with the same names outside the library. Since file name extensions have not been specified, default extensions are appended to the file names. The file names created for this example are MODA.OBJ, MODB.OBJ, and MODC.OBJ. File name extensions, listed in the *User's Guide*, are operating-system specific.

## FULLDIR — Displays Library or Library Module Contents

### Abbreviation

F

### Syntax

```
FULLDIR library_name [ (module_name, module_name)... ] [ list_filename ]
```

### Description

<i>library_name</i>	Specifies the library file whose contents are to be listed.  If you enter just the <i>library_name</i> , the contents of all modules are listed.
<i>module_name</i>	Specifies the module whose contents are to be listed.  When you enter specific <i>module_names</i> , information is displayed for the named modules only.
<i>list_filename</i>	Writes the directory display information to the named file. If not specified, the output defaults to the standard list device, usually the terminal.

The FULLDIR command provides a full directory display of a library's contents including module names, their sizes, and all external symbol definitions and references. The sizes listed are the number of bytes required to store the modules on the host computer system.

### Related Commands

DIRECTORY

**Example**

```
FULLDIR trig.lib (T1, T2) trig.lst
...
```

In the example above, the librarian displays information about modules T1 and T2 which are members of the library named TRIG.LIB. The output listing is written to the file named TRIG.LST.

```
Library trig.lib

Module      Size  Processor
T1 ...      414   68000

***** PUBLIC DEFINITIONS *****
vara
varb

***** EXTERNAL REFERENCES *****
var1
var2
var3

Public Count = 2
External Count = 3

Module      Size  Processor
T2 ...      785   68000

***** PUBLIC DEFINITIONS *****
pub1

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

## HELP — Displays Context-Sensitive Command Syntax

### Abbreviation

H

### Syntax

HELP

### Description

The **HELP** command lists commands with their correct invocation syntax. **HELP** is context-sensitive. The commands displayed are only those that can be legally entered at the time you type **HELP**.

### Related Commands

None

### Example

```
lib68k> help
CLEAR
CREATE library_name
DIRECTORY library_name[(module_name[,...])] [list_filename]
END
FULLDIR library_name[(module_name[,...])] [list_filename]
HELP
OPEN library_name
SAVE

lib68k> open rp005.lib

lib68k> help
ADDLIB library_name[(module_name[,...])]
ADDMOD filename[,...]
CLEAR
DELETE module_name[...]
DIRECTORY library_name[(module_name[,...])] [list_filename]
END
EXTRACT module_name[...]
FULLDIR library_name[(module_name[,...])] [list_filename]
HELP
REPLACE filename[,filename]
SAVE

lib68k> end
```

## OPEN — Opens an Existing Library

### Abbreviation

OP

### Syntax

OPEN *library\_name*

### Description

*library\_name*      Specifies the name of the library file to be opened.

The OPEN command lets an existing library be referenced in conjunction with succeeding commands that add, delete, replace, or extract modules. Only one library can be opened at a time.

If you create a new version of the library, the 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. In batch mode or command line entry, execution is terminated.

### Related Commands

CREATE

### Example

OPEN MATH.LIB

...

In the example above, the library named MATH.LIB is opened.

## REPLACE — Replaces Library Module

### Abbreviation

R

### Syntax

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

### Description

*filename*      Specifies a file containing one or more modules that will replace the module of the same name in the library named in the OPEN or CREATE command.

The REPLACE command replaces one or more library modules with one or more nonlibrary object modules with the same name. The replacement object module must have the same name as the library module it replaces.

REPLACE is not the same as DELETE mod1 followed by ADDMOD mod1 because ADDMOD always puts the new module at the end of the library whereas REPLACE retains the original module order. If the module does not already exist in the library, a warning is issued and the module is appended to the end of the library.

REPLACE must be preceded by an OPEN or CREATE command.

### Related Commands

ADDMOD, CREATE, DELETE, OPEN

### Example

```
OPEN    LIBRARY1.LIB
REPLACE SENTIN.OBJ,MODU1.OBJ
SAVE
...
```

In the example above, the OPEN command opens the library LIBRARY1.LIB. The library modules SENTIN.OBJ and MODU1.OBJ are replaced by nonlibrary modules of the same name.

**SAVE — Saves Contents of Current Library****Abbreviation**

S

**Syntax**

SAVE

**Description**

The **SAVE** command saves the contents of the library being created or modified. During this time, **ADDMOD**, **ADDLIB**, **DELETE**, and **REPLACE** commands are processed. Although these commands were checked for correct syntax and module existence at the time they were entered, the specified modules are not added, deleted, or replaced until a **SAVE** command is issued.

**Related Commands**

END

**Example**

```
CREATE NEW.LIB
ADDMOD REL1.OBJ, REL2.OBJ
ADDMOD FORTUN.OBJ
SAVE
...
```

In this example above, **NEW.LIB** is a newly created library comprised of the relocatable object modules named **REL1.OBJ**, **REL2.OBJ**, and **FORTUN.OBJ**.



# Sample Librarian Session 14

## Overview

The example librarian test programs and output listings in this chapter describe the input command file and the output listing format.

The sample test programs in this chapter use command files and object module files. Figure 14-1 shows the output listing for the first librarian sample program. Figure 14-2 shows the output listing for the second sample program.

During interactive program execution, the information is displayed at the terminal. When the librarian is executed in batch mode, the information is displayed in an output stream formatted like the output listings shown.

## Librarian Sample Program 1

The sample program 1 output listing is shown in Figure 14-1. The output listing contains the following information:

1. The command file is listed.
2. A new library, libtest1.lib, is created based on the **CREATE** command in the command file.
3. Two modules, lib68ka.obj and lib68ka.obj are added to this library.
4. The contents of the library are then listed with the **fulldir** command.
5. Each module name, the module's public definitions, and external references are listed.
6. The total public symbol count and external symbol count are listed for each module.
7. Total module count as well as total warnings and errors if any are displayed at the end of the listing.

```
Microtec Research LIB68K   Wed Jul 22 14:30:10 1992
Version x.y

* This test adds modules into a new library and lists
* them out to confirm correctness.

1 → cr test1.lib
addmod lib68ka.obj
addmod lib68kb.obj
list test1.lib
Microtec Research LIB68K      V x.y Wed Jul 22 14:30:10 1992

2 → Library being built test1.lib

Module      Size Processor
moda ...    387   68000
***** PUBLIC DEFINITIONS *****
moda_nine
moda_ten

***** EXTERNAL REFERENCES *****
moda_five
moda_two
moda_four
moda_three
moda_seven
moda_six
moda_eight
moda_one

3, 4, 5 → Public Count = 2
External Count = 8

Module      Size Processor
modb ...    387   68000
***** PUBLIC DEFINITIONS *****
modb_nine
modb_ten
```

Figure 14-1. Librarian Sample Program 1 Output Listing

\*\*\*\*\* EXTERNAL REFERENCES \*\*\*\*\*

modb\_five  
modb\_two  
modb\_four  
modb\_three  
modb\_seven  
modb\_six  
modb\_eight  
modb\_one

Public Count = 2  
External Count = 8  
Module Count = 2  
end

7 →

Figure 14-1. Librarian Sample Program 1 Output Listing (cont.)

## Librarian Sample Program 2

The sample program 2 output listing is shown in Figure 14-2. The output listing contains the following information:

1. The command file is listed with error or warning messages following any commands that could not be executed.
2. A new library, libtest2.lib, is created based on the **CREATE** command in the command file.
3. Two modules, lib68ka.obj and lib68kb.obj are added to this library.
4. A replacement is attempted on library module lib68kd.obj. However, the module is not in the library, so a warning message is issued indicating the module was not found. The module is then added to the library.
5. The contents of the library are then listed with the fulldir command.
6. The output listing shows each module name, the module's public definitions, and external references.
7. The total public symbol count and external symbol count are listed for each module.
8. The total module count as well as total warnings and errors if any are displayed at the end of the listing.

```
Microtec Research LIB66K      Wed Jul 22 14:36:14 1992
Version x.y

* create a library and add three modules trying to replace one with the
* one that doesn't have the same name of any present.
* This should cause an error or warning.

1 → create test2.lib
addmod lib66ka.obj
addmod lib66kb.obj
replace lib66kd.obj
        (201) Module modd not found.
        WARNING: (211) Module modd added.
fulldir test2.lib
Microtec Research LIB66K      V x.y Wed Jul 22 14:36:14 1992

2 → Library being built test2.lib

Module      Size Processor
moda ...     387   68000

3, 5, 6 → ***** PUBLIC DEFINITIONS *****
moda_nine
moda_ten

***** EXTERNAL REFERENCES *****
moda_five
moda_two
moda_four
moda_three
moda_seven
moda_six
moda_eight
moda_one

Public Count = 2
External Count = 8

Module      Size Processor
modb ...     387   68000
```

Figure 14-2. Librarian Sample Program 2 Output Listing

```
Module      Size Processor
modb ...    387   68000

***** PUBLIC DEFINITIONS *****
modb_nine
modb_ten

***** EXTERNAL REFERENCES *****
modb_five
modb_two
modb_four
modb_three
modb_seven
modb_six
modb_eight
modb_one

Public Count = 2
External Count = 8

Module      Size Processor
modd ...    387   68000

***** PUBLIC DEFINITIONS *****
modd_nine
modd_ten

***** EXTERNAL REFERENCES *****
modd_five
modd_two
modd_four
modd_three
modd_seven
modd_six
modd_eight
modd_one

Public Count = 2
External Count = 8
Module Count = 3
end

Warnings = 1
Errors   = 0
```

8 →

Figure 14-2. Librarian Sample Program 2 Output Listing (cont.)

# ASCII and EBCDIC Codes: Appendix A

The assembler will recognize the characters listed in Table A-1. The equivalent ASCII and EBCDIC codes are expressed in hexadecimal notation.

Table A-1. ASCII and EBCDIC Codes

Character	ASCII	EBCDIC	Character	ASCII	EBCDIC
blank	20	40	:	3A	7A
!	21	5A	;	3B	5E
"	22	7F	<	3C	4C
#	23	7B	=	3D	7E
\$	24	5B	>	3E	6E
%	25	6C	?	3F	6F
&	26	50	@	40	7C
'	27	7D	A	41	C1
(	28	4D	B	42	C2
)	29	5D	C	43	C3
*	2A	5C	D	44	C4
+	2B	4E	E	45	C5
,	2C	6B	F	46	C6
-	2D	60	G	47	C7
.	2E	4B	H	48	C8
/	2F	61	I	49	C9
0	30	F0	J	4A	D1
1	31	F1	K	4B	D2
2	32	F2	L	4C	D3
3	33	F3	M	4D	D4
4	34	F4	N	4E	D5
5	35	F5	O	4F	D6
6	36	F6	P	50	D7
7	37	F7	Q	51	D8
8	38	F8	R	52	D9
9	39	F9	S	53	E2

(cont.)

Table A-1. ASCII and EBCDIC Codes (cont.)

Character	ASCII	EBCDIC	Character	ASCII	EBCDIC
T	54	E3	j	6A	91
U	55	E4	k	6B	92
V	56	E5	l	6C	93
W	57	E6	m	6D	94
X	58	E7	n	6E	95
Y	59	E8	o	6F	96
Z	5A	E9	p	70	97
[	5B	C0	q	71	98
\	5C	E0	r	72	99
]	5E	D0	s	73	A2
^	5E	4A	t	74	A3
-	5F	6D	u	75	A4
`	60	79	v	76	A5
a	61	81	w	77	A6
b	62	82	x	78	A7
c	63	83	y	79	A8
d	64	84	z	7A	A9
e	65	85	{	7B	8B
f	66	86		7C	FA
g	67	87	}	7D	9B
h	68	88	~	7E	A1
i	69	89			

# Assembler Error Messages: Appendix B

## Introduction

This appendix describes the error messages and warnings that appear if errors in the source program are detected during the assembly process. The error message is printed on the listing immediately following the statement in error.

When the assembler finds a syntax error, it does not generate code for the instruction or directive on the line or for any of its continuation lines where the error occurs. The error message is printed on the line below the error with a caret pointing to the offending syntax. In some cases, the assembler issues a general syntax error that indicates there is something wrong at the place the caret points, but the specific nature of the error is not determined. It then continues processing with the next statement.

In the event of a syntax error, the assembler does not generate code, but it continues processing with the next statement.

The next section lists assembler messages with a description. Most error messages are self-explanatory. In all cases, unless otherwise noted, they represent error conditions that should be fixed before proceeding to the linker. Warning messages should be checked to verify that the assembler made the right assumptions prior to linking.

### Note

Messages that are outside the scope of this product (e.g., operating system error messages) are not documented in this appendix. Refer to the *Release Notes* for additional information on error messages.

If you encounter an undocumented error message, please contact Technical Support with the exact text of the message and the circumstances under which it occurred. The Technical Support representative will assist you in determining the cause of the problem.

## Assembler Messages and Errors

### 500 No error

### 501 Missing argument

The argument in a directive is missing or contains an illegal character.

### 502 Operator expected but not found

### 503 A symbol was found which is invalid in this context

A register name or a symbol is found when it is not expected. You may have specified the wrong keyword for a directive. Check the *Microtec Research ASM68K Reference Manual* for the correct symbol or character.

### 504 Right parenthesis not valid in this context

### 505 Operator not valid in this context

### 506 Expression terminator found prematurely

An expression terminator such as an end-of-line was found. Remove the terminator to fix the error.

### 507 Operand expected but not found

### 508 Unbalanced parentheses

The number of left-parentheses in an expression does not match the number of right-parentheses. The assembler ignores the line containing the expression and prints out an error message.

### 509 Complex relocatable value not valid in this context

Complex relocation was used in the EQU or ASSIGN directive. However, the assembler does not support complex relocation for these directives. Therefore, the expression whose value is assigned to the given label in each of these directives is ignored.

**510 Stack underflow (internal error)****511 Invalid operands for \ operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**512 Invalid operands for & operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**513 Invalid operands for | operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**514 Invalid operands for || operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**515 Invalid operands for = operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**516 Invalid operands for <> operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**517 Invalid operands for >= operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**518 Invalid operands for > operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**519 Invalid operands for < operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**520 Invalid operands for <= operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**521 Invalid operands for >> operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**522 Invalid operands for << operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**523 Invalid operands for \* operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**524 Invalid operands for / operator**

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

**525 Invalid character**

This message occurs as a result of syntactic errors. For example, a character may be invalid within the context where it occurs; an input line may be too long; or a register name may occur where it is illegal.

**526 Closing string delimiter missing****527 String longer than 4 characters invalid in this context**

A string of more than 4 characters was used as an operand for a 32-bit, 16-bit, or 8-bit value. Reduce the string to a maximum of 4 characters to fix the error.

**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 it 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 macro call. A macro defined within another macro, or conditional assembly statements is nested too deeply. **ELSEC**, **ENDC**, or **ENDM** has been used without a preceding **IF** or **MACRO**.

**533 Missing ENDC****534 IF stack overflow; limit is 16 nesting levels**

The assembler allows a maximum of 16 nesting levels for the **IF** conditional directive. Fix the error by reducing the number of nested **IF**s.

**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**

An integer value accompanying an instruction operand or directive operand was found which was outside the range for that particular instruction or directive. Check the *Microtec Research ASM68K Reference Manual* or the manufacturer's manual to determine the legal range.

**538 Label required on this directive**

The instruction contained an operand that violated a rule of relocation. Specifically, an operand that should have been absolute was being used as a relocatable expression. This error can also be due to an **ORG** directive making a reference to an external symbol. Fix the error by making the operand an absolute expression.

**539 Duplicate IDNT directive (ignored)****540 Relocatable expression invalid in this context**

One of the following errors may cause generation of this message: a relocatable expression is used for a field that is not 16 or 32 bits long; an operand that should be absolute is relocatable; or an **ORG** directive makes a reference to an external symbol.

**541 Comma expected but not found****542 Invalid section name**

A section name did not follow the rules for an assembler label, was a previously defined symbol, or the **SIZEOF** operator did not directly encounter a section name. Check the *Microtec Research ASM68K Reference Manual* for the rules governing the naming of a section.

**543 Section cannot be both COMMON and non-COMMON****544 Nested macro definition**

Nested macro definitions are not supported by the assembler. Fix the error by removing the nesting.

**545 Too many sections**

A maximum of 30,000 sections including absolute sections are allowed. The **SECTION** directive is ignored. Fix the error by reducing the number of sections or by splitting up the assembly source file.

**546 Invalid symbol**

**547 This sort of symbol cannot be made an external definition**

The symbol has already been declared a section, a module name, an external, a register, or the symbol has been defined by the **ASSIGN** directive. Fix the error by not allowing the symbol to have an external declaration.

**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**

The rules surrounding the definition of formal parameters for a macro have been violated. Check the *Microtec Research ASM68K Reference Manual* for the correct method of defining a 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 **XREF** 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**

You have specified an option for the directive which the assembler cannot recognize. Check the *Microtec Research ASM68K Reference Manual* for a list of options that the directive allows.

**558 Register list invalid in this context**

A register list has been specified as an operand of a directive or instruction. The register list is invalid for this operand. Fix the error by verifying that the operands of the instruction or directive are correct or by verifying the use of the register list in a register range.

**559 .W or .L extension on register not valid in this context**

You should verify the validity of the extension on the 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. This message indicates improper use of the registers in this instance.

**561 A colon-separated pair of registers is invalid in this context**

Register pairs cannot be separated by a colon in this instruction. This message indicates improper use of the colon in the syntax.

**562 Register expected but not found**

A register was expected in an instruction operand. Replace the incorrect operand with a register or register expression.

**563 A register in a register list is invalid in this context**

An invalid register has been specified in the register list. This may occur in the REG directive when a floating-point register is specified or vice versa. Fix the error by verifying that the register list specified is valid.

**564 Registers separated by - in register list must be in ascending order**

A register range has been specified in which the first register number is greater than the second register number. Fix the error by ensuring that the first register is a smaller numbered register than the second register.

**565 Registers separated by - in register list must be of same type****566 Invalid expression containing 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**
- 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**

Invalid extension for nested 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 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**

Condition code required for this instruction.

**585 Code generated is unequivalent in some cases. Recoding recommended****586 THEN expected but not found****587 This instruction has too many operands**

The assembler has found a mnemonic which has too many operands. Check the *Microtec Research ASM68K Reference Manual* to determine the number of operands allowed for the particular instruction, and fix the error.

**588 This combination of operands is not valid for this instruction**

An invalid addressing mode has been found in one of the operands. Check the *Microtec Research ASM68K Reference Manual* to determine the addressing mode allowed for the particular mnemonic, and fix the error.

**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 generated program 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**

An invalid character was found while the assembler was reading a floating-point constant. Check the *Microtec Research ASM68K Reference Manual* for the definition of a floating-point constant.

**595 A real constant was expected here**

Floating-point directives and floating-point instructions require floating-point constants as operands or parameters.

**596 Real numbers invalid in this context**

Floating-point constants can only be used as operands for floating-point directives and floating-point instructions. They cannot be used within arithmetic expressions.

**597 This real number is too small to represent. Zero substituted**

This is only a warning.

**598 This real number is too large to represent. Infinity substituted**

This is only a warning.

**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 processor instruction.

**602 Calculated displacement does not fit in its field - truncated****603 Structured Directives not properly closed****604 Maximum number of typed sections exceeded in HP mode**

When assembling with the **-h** option, more than one relocatable section was mapped to HP section **PROG**, to HP section **DATA**, or to HP section **COMN**. Local

symbols from these extra sections are not written to the **asmb\_sym** file and will be unavailable for debugging. To eliminate this warning, move the extra sections into a new source module.

**605 Out of virtual memory**

No further virtual memory space is available for use.

**606 Invalid Value for alignment, can only be 0, 1, 2 or 4****607 End of File inside a macro or repeat definition****608 Expression stack overflow**

Expressions are stored in a stack to facilitate evaluation. The stack can be exceeded by a very long expression.

**609 Value is outside of its legal range****610 Illegal branch to odd address****611 Unable to create or open an 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****621 Macro/repeat definition terminated by assembler.**

An **ENDR** (end of repeat) and **ENDM** (end of macro) is provided by the assembler in order to terminate an incomplete macro/repeat definition. This is a warning.

**623 Too many formal parameters. Limit is 36**

You have too many formal parameters in a macro definition. Fix the error by reducing the number of formal parameters.

**624 Macro names cannot contain a period (.)**

A period can only appear as the first character of a macro name. Other than this exception, it will be considered as part of a qualifier (.B, .W, .L, or .S) when the macro is called.

**625 Macro definition has too many local symbols**

You have too many local symbols in a macro definition. The maximum number of local symbols allowed in a macro definition is 90. Reduce the number of local symbols in your macro definition.

**626 Invalid model parameter**

The model parameter may be missing in the IRP assembler directive. This is an error.

**627 Expanded macro line is too long**

The line in the macro definition is too long for the macro preprocessor's internal buffer. This is an error. You should break the line into two shorter ones, and reassemble.

**629 Illegal CHIP identifier****630 Invalid operand for .STARTOF. operator**

A section name is expected as an operand.

**631 Invalid operand for .SIZEOF. operator**

A section name is expected as an operand.

**632 The number of nesting levels for macros cannot exceed the maximum**

The maximum nesting level is 8 (PC hosts) or 100 (non-PC hosts).

**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**

Extra operands were encountered on the remaining source lines and were ignored.

**635 OPNOP option is only allowed on the command line. Option ignored**

This option must be on or off throughout the entire assembly. Therefore, it cannot be specified in the LIST directive which is position dependent in the source file.

# Linker Error Messages: Appendix C

## Introduction

This appendix describes the error messages and warnings that can appear during linking. Errors and messages are listed beneath the actual command in error. For most load errors, 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.

Load messages normally occur during the loading of object modules initiated by the **LOAD** command. Errors and messages from the linker will be nonfatal or fatal. If the error is nonfatal, the load will proceed after the error is reported. If the error is fatal, the linker will report the error, and the load will terminate immediately.

### Note

Messages that are outside the scope of this product (e.g., operating system error messages) are not documented in this appendix. Refer to the *Release Notes* for additional information on error messages.

If you encounter an undocumented error message, please contact Technical Support with the exact text of the message and the circumstances under which it occurred. The Technical Support representative will assist you in determining the cause of the problem.

## Message Severity Levels

There are two classes of errors that can occur during linker program execution. The first class is nonfatal and processing proceeds after the error is reported. The second class is fatal and processing is abandoned.

Messages can be one of the four types listed in Table C-1.

Table C-1. Linker Message Severity Levels

Type	Severity
Warnings	Not fatal.
Messages	Informative and often appears in conjunction with another error message.
Errors	Usually fatal, but processing continues to facilitate further error checking.
Fatal	Always fatal. Processing aborts.

## Linker Messages and Errors

### 300 Bad IEEE Object Record

Either the object module has been corrupted or it is not a Microtec Research IEEE-695 relocatable object file.

### 301 Maximum Number Of Sections Exceeded

The maximum number of allowable sections (30,000) 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 newly found sections.

### 303 Section Overlap

Due to addresses that you have specified or absolute sections, one or more of the sections overlap. This is an informative message and loading continues.

### 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**

The linker has run out of memory in the host system.

**306 Out of memory**

The linker 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.

**308 Invalid CHIP Command**

The CHIP command you specified is not a legal linker command.

**309 Invalid Command**

You have specified a linker command that is not legal.

**310 Load Completed**

Message indicates normal load.

**311 Load Not Completed**

Message indicates abnormal load.

**312 Invalid ORDER command**

You have specified an ORDER command in such a way that it is not a legal linker command. Check the syntax for this command in the *Microtec Research ASM68K Reference Manual*.

**313 Invalid Operand**

An operand specified for a command contains invalid characters, does not exist, or is too large.

**314 Chip inconsistency**

The linker has encountered a file assembled with a **CHIP** directive which has greater capabilities than the **CHIP** specified to the linker. For example, a file assembled with the **CHIP 68020** directive is loaded with the **CHIP 68000** link command in effect. The module may contain instructions which cannot execute on the target chip.

**315 Maximum memory exceeded**

The program exceeds the memory available for the target microprocessor.

**316 Short memory 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 linker has encountered a fatal internal error. A seek error can be caused by a device that is full.

**319 Cannot Open File**

The linker is unable to open the relocatable object file.

**320 UNRESOLVED EXTERNALS**

The unresolved external symbols are listed following this error message.

**322 8-bit 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. You 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 \$FFFFFF or 0 to \$7F) displacements that will be sign-extended to 32 bits at runtime (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 \$FFFFFF.

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 link 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 link map.

### 323 16-bit 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 continues, but the program may not run properly. The location *nnnn* indicates the offset within section *yyyy* of module *xxxx*. You should investigate.

For example, an absolute short instruction refers to a location that is not in the range \$0 through \$7FFF or \$FFFF8000 through \$FFFFFF. 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 linker 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 runtime, then the value must fall within the range \$0 through \$7FFF or \$FFFF8000 through \$FFFFFF. If the field will not be sign-extended, the value must fall in the range \$0 through \$FFF or \$FFF0000 through \$FFFFFF.

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 for Symbol

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 **XDEF**s as well.

**325 Illegal HP section name**

The HP object file contains an illegal section name.

**326 Cannot open temporary file**

The CONFIG.SYS file may not have had enough files and buffers. For more information, refer to the *Microtec Research ASM68K Installation Guide*.

**327 Illegal ALIAS command**

The ALIAS command was used illegally in the code.

**328 Illegal command for an ALIAS section**

A section that was aliased via ALIAS to another section was mentioned in a linker 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 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 linker has encountered a relocatable module that it cannot properly interpret. Usually, this results from using different versions of assembler and linker programs. A later version of the assembler will produce a relocatable object module that is rejected by an earlier version of the linker.

**332 Object contains errors**

The assembler detected errors when the relocatable object was produced. Check the source code for instructions that will not execute properly.

**333 Source file does not exist**

This message is for debugging purposes. The debugger normally issues an informative message containing the location of the source file. If the source was compiled and assembled and then the source file was deleted, the linker issues this informative message.

**334 Local symbols in CODE section**

Local symbols were encountered in the **CODE** section. Local symbols are illegal in the **CODE** section.

**335 Local symbols in DATA section**

Local symbols were encountered in the **DATA** section. Local symbols are illegal in the **DATA** section.

**336 Local symbols in COMN section**

Local symbols were encountered in the **COMN** section. Local symbols are illegal in the **COMN** section.

**337 Illegal command for incremental linking**

The command file can contain only **LOAD** statements when incremental linking is used. Any other statements in the command file will generate this error.

**338 Duplicate ROM section**

Duplicate INTTADATA message used in older versions of the linker.

**339 Section moved to high short section**

As the linker 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. High base page depends on the **CHIP** command, as follows:

68008	\$000F8000	through \$000FFFFF
68000/10	\$00FF8000	through \$00FFFFFF
68020/30/40	\$FFFF8000	through \$FFFFFFFF

**340 Out of virtual memory**

No further virtual memory space is available for use.

**341 This command is illegal after LOAD is used**

An illegal command has been used following a **LOAD** command.

**342 Incompatible incrementally linked object. Recreate the object**

The object file was incrementally linked by a Version 6.4b or earlier version of the linker. This object file is not compatible with Microtec Research's latest version of the linker. Relink using the latest version of Microtec Research's linker.

**345 Duplicate Public From Library Module -- ignored**

The public symbol that caused this message to appear was already defined in the library. Loading will continue, and the symbol will be listed on the link map. This message is a warning.

**346 Could Not Construct Full Path Name**

You may have tried to link objects that were created on a different host. Re-create the objects on the same host and then relink. This message indicates a nonfatal error.

**347 Command Ignored:**

This warning message usually appears with a companion message that gives the reason for why the command was ignored.

**348 Module Not Found.**

This message indicates a nonfatal error.

**349 Section Previously Specified or Non-existent**

A section or subsection specified by the **MERGE** command is either non-existent or already merged. This message is a warning.

**350 Illegal Multiple Case Specification for *class***

Each class (**PUBLICS**, **MODULES**, **SECTIONS**) can have only one case specification (**CASE**, **UPPERCASE**, or **LOWERCASE**). If more than one case specification is used for *class*, this error message is generated. This message indicates a nonfatal error.

**351 Write error - disk may be full.****352 Section Mismatch Between PUBLIC Def and Module Ref for Symbol**

A section mismatch has occurred for a symbol that was defined by a **PUBLIC** command and referenced in some module by an **XREF** assembler directive. This message is a warning.

**353 Redefinition of *symbol\_name***

A *symbol\_name* defined by the **PUBLIC** command or a register has been redefined. A register can be (re)defined using the **INDEX** command. A public symbol can be (re)defined using the **PUBLIC** linker command or the **XDEF** assembler directive. The value of the symbol specified by the last **PUBLIC** command will override the previous value. This message is a warning.

**354 Cannot initialize buffer for IEEE I/O****355 MERGE and ALIAS cannot be used together****356 Section not found, *section*****362 Too Many Errors**

Too many errors have been found. Any additional errors found after this message is shown will not be reported.

**364 Cannot ABSOLUTE unknown section, *section***

The section is not defined in any of the modules loaded by the linker.

**365 Cannot ALIGN unknown section, *section***

The section is not defined in any of the modules loaded by the linker.

**366 Cannot ALIGN absolute section, *section***

Absolute sections have a fixed starting address and cannot be aligned.

**367 Absolute section cannot have the same name as other sections, *section***

Sections with a defined location address cannot be combined with a relocatable section. This error message appears when an absolute section has the same name as a relocatable section.

**368 Combined section exceeds memory space, *section***

The combined section has a size greater than the processor memory space. The program's code or data is too large as a result.

**372 Section size shrunk for *section***

The default size of a section is greater than the size specified by the SECTSIZE command. Only sections that are of type common can be shrunk.

**373 24-bit Value Out of Range at *address* in module *section* *section***

The relocated value of an expression will not fit into a 24-bit field. The address *address* indicates the offset within the named section of the named module. Loading continues, but the program may not run properly; you should investigate.

All expressions are evaluated as unsigned 32-bit values. These unsigned values are to be within the range of sign-extended 24-bit values (i.e., \$FF800000 to \$FFFFFF or 0 to \$7FFFFFF). At runtime, these values will be sign-extended to 32 bits. In the more common case of immediate values which are not sign-extended at runtime, the expected range is 0 to \$FFFFFF.

In either case, the value inserted in the object module is the low 24 bits of the complete 32-bit value. This message interrupts the link 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 map file.

**374 ORDER command could not be obeyed for section, *section***

The linker is unable to allocate memory in the order specified for the section listed in the **ORDER** command. A possible example is:

```
ORDER sect1,sect2,sect3  
SECT sect2=0
```

Since **sect2** must begin at address 0, there is no way **sect1** can precede it.

**375 No modules were loaded**

No **LOAD** or **LOAD\_SYMBOLS** commands were specified.

The linker assumes that something is wrong if no modules were actually loaded. There may be a syntax error preventing the **LOAD** command from being read, such as the **END** command inserted before any **LOAD** commands. If the only **LOAD** argument is a library file, the linker will load modules from a library only if they resolve undefined externals. If there are no undefined externals, the linker will not **LOAD** the library.

**376 Invalid modifier, *modifier***

The linker command does not support this modifier. This message is a warning. For legal modifiers, refer to *Chapter 10, Linker Commands* in this manual.

**377 Duplicate section name specified in INITDATA command(s)**

The **INITDATA** command contains duplicate section names. This message is a warning.

**379 Invalid INITDATA command**

Missing operands in the **INITDATA** command.

**381 Definition of public symbol is ignored**

Symbol has already been defined in one of the incoming object modules.

**384 Missing operand**

A linker command requires at least one additional operand.



# Librarian Error Messages: Appendix D

---

## Introduction

This appendix describes the error messages and warnings that appear if errors are detected while running the librarian. The error message is printed on the listing immediately following the statement in error.

### Note

Messages that are outside the scope of this product (e.g., operating system error messages) are not documented in this appendix. Refer to the *Release Notes* for additional information on error messages.

If you encounter an undocumented error message, please contact Technical Support with the exact text of the message and the circumstances under which it occurred. The Technical Support representative will assist you in determining the cause of the problem.

## Message Severity Levels

There are two classes of errors that can occur during program execution of the librarian. The first class is nonfatal and processing proceeds after the error is reported. The second class is fatal and processing is abandoned.

Messages are of one of the types shown in Table D-1.

Table D-1. Librarian Message Severity Levels

Type	Severity
Warnings	Not fatal.
Messages	Informative and often appears in conjunction with another error message.
Errors	Usually fatal, but processing continues to facilitate further error checking.
Fatal	Always fatal. Processing aborts.

## Librarian Error Messages

### 100 Could not close file *filename* to open another file.

In order to reduce processing overhead, the librarian keeps files open with the OPEN command. This message is displayed if too many files are open and the librarian unsuccessfully attempts to close a file in order to open a new one. To remedy this situation, reduce the number of files you are working with during a given session.

### 101 Unable to open file *filename*.

The librarian could not open the named file when executing an ADDMOD, REPLACE, or OPEN command. This error could be caused by either an invalid file name specification or when the specified file does not exist. The librarian ignores the command that generates this error.

### 102 Unable to close file *filename*.

The librarian generates this error when it encounters an operating system error and cannot close the named file. This message typically is accompanied by another error message that provides a more specific reason for not closing the named file.

### 104 File *filename* not included.

The librarian issues this message when it cannot execute the ADDMOD command because the named file is corrupted or does not exist. This message has a companion message that specifically states why the named file is not included in the library.

**106 File *library\_name* exists already.**

The librarian generates this message when you use the **CREATE** command and the named library currently exists. The librarian displays a warning in batch and interactive modes.

**107 File *filename* does not exist.**

The librarian generates this message when you issue an **OPEN** command and the named file does not exist.

**108 Library file *library\_name* not opened.**

This message has a companion message that specifically states why the library file was not opened.

**109 Library file *library\_name* not opened.**

This message has a companion message that specifically states why the library file was not opened.

**120 Use HELP for proper command syntax.**

This message suggests using the **HELP** librarian command to display correct syntax for all librarian commands that can be entered at this point in the session.

**201 Module *module\_name* not found.**

The librarian could not locate the named module in the library to execute a **DELETE**, **REPLACE**, or **EXTRACT** command.

**203 Module *module\_name* already exists in current library.**

The librarian cannot execute an **ADDMOD** or **ADDLIB** command because the module named in the message exists in the current library. If you wish to replace a module in the library, use the **REPLACE** command. The librarian ignores the **ADDMOD** or **ADDLIB** command that contains a duplicate module name.

**204 *filename* is a library file.**

The librarian generates this error message when it attempts to execute an **ADDMOD** or **OPEN** command, and the associated file name is not an object module. The command containing the erroneous file is ignored.

**205 *filename* is not a library file.**

The librarian issues this command when it attempts to execute an **ADDLIB** or **OPEN** command, and the associated file name is an object module. The librarian ignores the command containing the erroneous file.

**206 Module *module\_name* is not included in the library.**

The librarian issues this message with a companion message that gives the specific reason for not including the named module in the library. This messages describes the result: the named module is not included in the library.

**207 Bad object record.**

Either the object module has been corrupted or it is not a legal relocatable object file. The librarian issues this message with a companion message, which names the file with the bad object record. Whatever command is associated with the bad object record file will be ignored.

**208 Bad library header record.**

The library has a bad header record. The librarian issues this message with a companion message which names the file with the bad header record. The command associated with the bad library header record will be ignored.

**209 Duplicate symbol *symbol\_name*.**

A module named in an **ADDLIB**, **ADDMOD**, or **REPLACE** command has the same public definition symbol that occurs in another module. The librarian issues this message with a companion message that provides information about what action it takes.

The librarian considers symbols to be case-sensitive.

**210 Bad object record in file *filename*.**

The named library or module file may have been corrupted.

**250 Out of memory.**

The librarian issues this message when it encounters insufficient system memory to execute commands issued since the last **CREATE** or **OPEN** command.

**251 Failed writing library. *Reason*.**

The librarian generates this message when it attempts to execute a **SAVE** command, and cannot. The message provides the reason for the inability to create a library. The librarian abandons the current session affected by the **SAVE** command that caused the error.

**253 Library *library\_name* not written.**

The librarian issues this message when an error occurs earlier in the session that prevents the library from being saved. This message is typically accompanied by another message that contains the reason the named library was not created.

**254 Failed writing module *module\_name* to file *filename*.**

When attempting to execute an **EXTRACT** command, the librarian cannot write the named module from an existing library to the new file which is external to the library. A companion error message describes the reason that the module cannot be extracted. If an error is encountered in batch mode, all commands following the **EXTRACT** command will not be executed; however, they will still be checked for syntactical validity.

**255 Replacement not done.**

The librarian issues this message when it cannot execute the **REPLACE** command for the reason specified in the companion message.

**256 Extraction failed.**

The module named in the **EXTRACT** command is not extracted.

**257 Syntax error in command.**

The librarian generates this message when it encounters an incorrect command sequence or an incorrect command syntax. For example, if your batch file does not have a terminating END or QUIT command, this error is generated.

## Glossary: Appendix E

---

<b>Absolute Section</b>	Part of assembly program that is to be loaded at fixed locations in memory. It contains no relocatable information.
<b>Address Expression</b>	An expression whose value represents a location in memory.
<b>Base Address</b>	Lowest address considered for loading relocatable sections of the absolute object module.
<b>Common Section</b>	A section type. This type of section contains variables that can be referenced by each module. All common subsections are loaded beginning at the same address.
<b>Load Address</b>	Memory address where the lowest byte of a section is placed.
<b>Module</b>	A module is the relocatable object code resulting from a single assembly. It can contain pieces of one or more sections.
	The linker combines pieces of a section from different modules. Such pieces always make up a contiguous block of memory, assuming they can be combined at all.
<b>Noncommon Section</b>	A section type. This type of section contains code only. All subsections of a noncommon section are loaded into a contiguous block of memory and do not overlap.
<b>Numeric Expression</b>	An expression whose value represents a number.
<b>Register Expression</b>	An address expression whose base or index attribute is nonnull.
<b>Relocatable Section</b>	General purpose section which can contain both instructions and/or data.
<b>Short Section</b>	A section type. This type of section can be referenced by absolute short addressing mode.
<b>Starting Address</b>	Location where execution begins.

**Subsection**

Individual pieces of code from various modules which make up a section.

# Object Module Formats: Appendix F

---

This appendix describes the Motorola S-record absolute object file format.

## S-Record Format

A file in Motorola S-record format is an ASCII file. Absolute object files consist of optional symbol table information, data specifications for loading memory, and a terminator record.

```
[ $$ [module_record]
symbol records
$$ [module_record]
symbol records
$$ ]
header record
data records
record count record
terminator record
```

### Module Record (Optional)

Each object file contains one record for each module that is a component of it. This record contains the name of the module. There is one module record for each relocatable object created by the assembler. The name of the relocatable object module contained in the record comes from the IDNT directive. For absolute objects created by the linker, there is one module record for each relocatable object file linked, plus an additional record for the linker whose name comes from the NAME command.

#### Example:

```
$$ MODNAME
```

### Symbol Record (Optional)

As many symbol records as needed can be contained in the object module. Up to 4 symbols per line can be used, but it is not mandatory that each line contain 4 symbols. A module can contain only symbol records.

**Example:**

```
APPLE $00000 LABEL1 $0D0C3
MEM $0FFFF ZEEK $01947
```

The module name associated with the symbols can be specified in the *module\_record* preceding the symbol records.

**Example:**

```
$$MAIN
```

Symbols are assumed to be in the module named in the preceding *module\_record* until another module is specified with another *module\_record*. Symbols defined by the linker's PUBLIC command appear following the first module record, which indicates the name of the output object module specified by the linker's NAME command.

**Header Record**

Each object module has exactly one header record with the following format:

```
S00600004844521B
```

**Description:**

S0	Identifies the record as a header record.
06	The number of bytes following this one.
0000	The address field, which is ignored.
484452	The string HDR in ASCII.
1B	The checksum.

**Data Record**

A data record specifies data bytes that are to be loaded into memory. The format for such a record is shown in Figure F-1. The columns shown in the figure represent half a byte (4 bits).

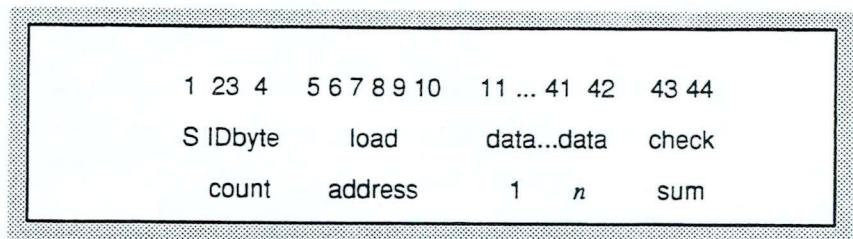


Figure F-1. Data Record Format for 16-Bit Load Address

Column	Description
1	Contains S which indicates the start of a record in Motorola S-record format.
2	Contains a digit identifying the record type. For data records, this digit is 2 (for 24-bit addresses) or 3 (for 32-bit addresses only if CHIP CPU32, CHIP 68020, CHIP 68030, or CHIP 68040 was specified).
3 to 4	Contains the count of the number of bytes following this one within the record. The count includes the checksum and the load address bytes but not the byte count itself.

For 24-bit load address:

- 5 to 10 Contains the load address. The first data byte is to be loaded into this address and subsequent bytes into the next sequential address. Columns 5 and 6 contain the high order address byte, columns 9 and 10 contain the low order address byte.
- 11 to 42 Contains the specifications for up to 16 bytes of data.

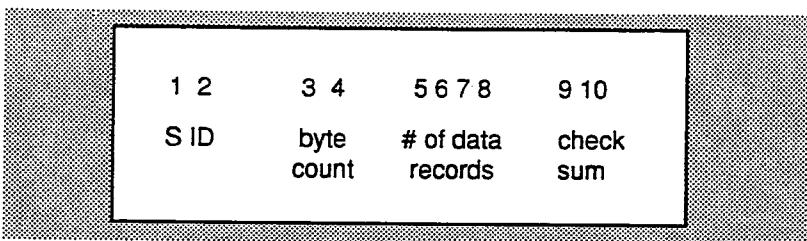
For 32-bit load address:

- 5 to 12 Contains the load address. The first data byte is to be loaded into this address and subsequent bytes into the next sequential address. Columns 5 and 6 contain the high-order address byte, columns 11 and 12 contain the low-order address byte.
- 13 to 42 Contains the specifications for up to 16 bytes of data.

The last two columns in the data record contain a checksum for the record. To calculate this, take the sum of the values of all bytes from the byte count up to the last data byte, inclusive, modulo 256. Subtract this result from 255.

**Record Count Record**

The record count record verifies the number of data records preceding it. The format for such a record is shown in Figure F-2. The columns shown in the figure represent half a byte (4 bits).



**Figure F-2. Record Count Record Format**

Column	Description
1	Contains S which indicates the start of a record in Motorola S-record format.
2	Contains 5 which indicates a record count record.
3 to 4	Contains the byte count 03.
5 to 8	Contains the number of data records in this file. The high-order byte is in columns 5 and 6).
9 to 10	Contains the checksum for the record.

**Example:**

S503010DEE

The example above shows a record count record indicating a total of 269 records (0x010D) and a checksum of 0xEE.

A terminator record specifies the end of the data records. The format for such a record is shown in Figure F-3.

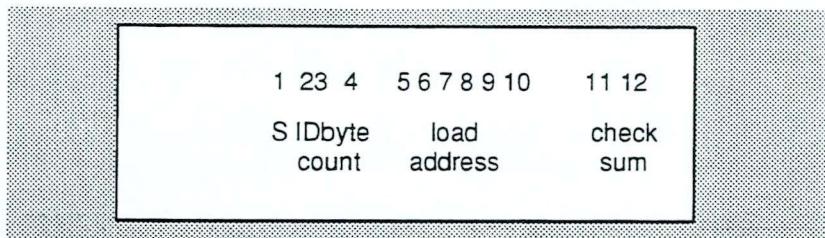


Figure F-3. Terminator Record Format for 24-Bit Load Address

Column	Description
1	Contains S which indicates the start of a record in Motorola S-Record format.
2	Contains 8 to indicate a 24-bit load address. It contains 7 for CHIP CPU32, CHIP 68020, CHIP 68030, or CHIP 68040 to indicate a 32-bit load address.
3 to 4	Contains the byte count 04.

For 24-bit load address:

5 to 10 Contains the load address which is either set to zero or to the starting address specified in the END directive (there are no data bytes).

11 to 12 Contains the checksum for the record.

For 32-bit load address:

5 to 12 Contains the load address which is either set to zero or to the starting address specified in the END directive (there are no data bytes).

13 to 14 Contains the checksum for the record.

**Example:**

S804000AF0001

The above example shows a terminator record with a 24-bit load address of 0x000AF0 and a checksum of 0x01.

## Sample S-Record

The following command file uses the format s linker command:

```
*  
* Test program for Linker  
*  
* This test program links three object modules from  
* three separate files. Commands are used to set the  
* entry point for the relocatable sections, set the  
* section load order, and to set the stack location  
*  
format s  
*  
listmap crossref,public,internals  
*listabs puts debug info in output file  
listabs publics,internals  
* order the relocatable sections  
order code,data  
public extraneous=$3000  
name testcase  
* set the base address for loading  
base $200  
*  
* set stack pointer  
public stack_top=$2000  
* load first two object modules  
load lnk68Ka,lnk68Kb  
* load last object module  
load lnk68Kc  
end
```

The resulting S-record output is as follows:



# C++ Support: Appendix G

## Overview

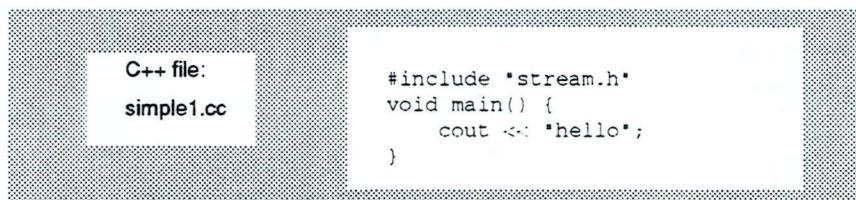
This appendix discusses modifications made to the ASM68K Assembler, LNK68K Linker, and LIB68K Librarian to support C++ name mangling and demangling.

## Name Mangling and Demangling

In the C++ language, you can use the same names to refer to different operations; this practice is known as "overloading." Since the same name is used, C++ introduced a unique naming encoding scheme to differentiate between two names used for the different operations. Therefore, you can name one function `closeout(char *name, float balance)` to indicate closing out a bank account, and pass it a parameter indicating the customer's name and return a parameter indicating the closing balance. You can name another function in your program `closeout(char *telex, int *account)` indicating a bank branch to notify about the customer's closed account. Although both these functions have the same name (`closeout`), C++ uniquely identifies each function based on the parameters passed by encoding this information into a "mangled" name.

## ASM68K Assembler

A sample C++ file, `simple1.cc`, was compiled on a UNIX operating system. The resulting `simple1.o` object file was linked with the `map` option set, so that a link map was generated. Figure G-1 shows the sample C++ source file.



The screenshot shows a terminal window with a black background and white text. On the left, there is a small text box containing the text "C++ file: simple1.cc". To the right of this box is the actual C++ source code:

```
#include <iostream.h>
void main() {
    cout << "Hello";
}
```

Figure G-1. C++ Program Listing

Use the **OPT CRE** or **OPT X** directive in your assembler source file to demangle all labels that fit the C++ name encoding scheme in your cross-reference and

symbol tables. Both the mangled and demangled C++ names (i.e., the original C++ source file name) will be listed for the cross-reference and symbol tables in the assembler output listing. The following example shows a portion of the symbol table from the assembler output file generated for `simple1.cc`:

#### **Example:**

Symbol Table	
Label	Value
<code>__S0_iostream_init</code>	<code>zerovars:00000000</code>
<code>__S1</code>	<code>strings :00000000</code>
<code>__ct_13Iostream_initFv</code>	
<code>Iostream_init:::Iostream_init()</code>	External
<code>__dt_13Iostream_initFv</code>	
<code>Iostream_init:::~Iostream_init()</code>	External
<code>__ls_7ostreamFPCC</code>	
<code>ostream::operator&lt;&lt;(const char*)</code>	External
<code>__std_simple1_cc_main_</code>	<code>code :0000002C</code>
<code>__sti_simple1_cc_main_</code>	<code>code :0000001C</code>
<code>__main</code>	External
<code>_adjustfield_3ios</code>	
<code>ios::adjustfield</code>	External

## **LNK68K Linker**

The LNK68K Linker also supports name C++ mangling and demangling. Since the mangled name is not easily interpreted, the decoded (or “demangled”) name is also shown on the linker error message output and map files. In the map file output, C++ mangled names can appear in the PUBLIC SYMBOL TABLE section. If a public linkage name fits the C++ name mangling pattern, LNK68K will report the C++ mangled name as well as the C++ demangled name in the map file.

#### **Demangling Example**

One encoded name of a function defined in the C++ I/O stream class library is:

```
__as_18istream_withassignFP9streambuf
```

LNK68K will decode the above encoded C++ name into the C++ symbolic source name:

```
istream_withassign::operator=(streambuf*)
```

Now, you can see that the class `istream_withassign` contains an overloaded operator function (=), which takes one argument of type `streambuf*`.

LNK68K will report the mangled name and the demangled name in the public symbol table entry of that function in the linker map file:

PUBLIC SYMBOL TABLE			
SYMBOL	SECTION	ADDRESS	MODULE
....			
<code>__as_18istream_withassignFP9streambuf</code>	code	000110DE	stream
<code>istream_withassign::operator=(streambuf*)</code>			
....			

The demangled name of the function is listed in the **SYMBOL** column of `__as_18istream_withassignFP9streambuf`.

## Symbol Name Length

LNK68K's linkage name limit has been extended to handle 127 characters, instead of the standard 31 characters. For symbols that are longer than the line limit, LNK68K will continue the symbol on the next line. The **SECTION**, **ADDRESS**, and **MODULE** information will start in the corresponding columns of the next line following the long symbol name.

A link map was generated for `simple1.cc` running on a UNIX-based host. The following list describes several C++ unique items that appeared in the link map:

1. The `initfini` section is a special C++ section reserved for pointers to C++ static constructors and destructors. For more information, refer to your C++ Compiler *Reference Manual*.

### Example:

SECTION SUMMARY						
-----						
SECTION	ATTRIBUTE	START	END	LENGTH	ALIGN	
initfini	NORMAL ROM	000100C8	000100FF	00000038	2	(WORD)

2. Mangled names have been decoded to show their demangled names. Symbolic names are listed in mangled or demangled pairs.

### Example:

```
__as_18istream_withassignFP9streambuf
istream_withassign::operator=(streambuf*)
```

3. Names prefixed by `__ptbl__` are the linkage names for pointers to the C++ virtual tables produced by the C++ compiler.

**Example:**

```
____ptbl____3ios____8iostream____stream_stream_cxx
```

4. Names prefixed by `__std__` are the linkage names for pointers to the C++ static destructors.

**Example:**

```
____std____stream_filebuf_cxx_last_op_
```

5. Names prefixed by `__sti__` are the linkage names for pointers to the C++ static constructors.

**Example:**

```
____sti____stream_filebuf_cxx_last_op_
```

6. Names prefixed by `__vtbl__` are the linkage names for pointers to the C++ virtual tables produced by the C++ compiler.

**Example:**

```
____vtbl____7ostream
```

## LIB68K Librarian

LIB68K will automatically demangle C++ names when they are encountered if a given symbol name fits the C++ name mangling scheme. If there is a C++ symbolic source-level name available for the symbol name being processed, LIB68K will display the C++ symbolic name in the next line of the low-level name:

```
<low-level mangled c++ name>
C++ NAME: <c++ demangled symbolic source-level name>
```

**Example:**

```
_flush____7ostreamFv
C++ NAME: ostream::flush()
```

In this example, the mangled name shows the function signature. The demangled name shows that there is a function `flush()` in the class `ostream` which does not take any arguments.

# Index

## Symbols

\$ 2-10, 2-12  
% 2-12  
( ) 13-1  
\* 5-11, 13-1, 13-2  
+ 7-14, 13-2  
, 13-1  
; 13-1, 13-2  
⟨⟩ (macro notation) 6-3  
== (exists) 6-3  
?? 2-7, 7-4  
# continuation character 9-11, 10-49, 10-59  
@ 2-12  
\ 6-2  
\@ 2-8

## A

A2-A5 relative addressing 3-24 to 3-33  
accessing dynamically allocated areas 3-27  
accessing statically allocated areas 3-25  
advantages 3-24  
Absolute expression 3-24, 4-9

memory indirect pre-indexed 3-11  
program counter memory indirect  
post-indexed 3-12  
pre-indexed 3-13  
program counter relative 3-8  
program counter with base  
displacement and index 3-12  
program counter with displacement 3-12  
register direct 3-9  
register indirect 3-9  
8-bit displacement and index 3-10  
base displacement and index 3-10  
displacement 3-9  
postincrement 3-9  
predecrement 3-9  
selection 3-20 to 3-22  
syntax 3-15 to 3-17  
user control 3-22 to 3-24  
Addressing modes  
(see Address modes)  
ALIAS linker command 10-9 to 10-10  
ALIGN assembler directive 5-4  
ALIGN linker command 10-11 to 10-12  
Alignment, sections 9-6  
ALIGNMOD linker command 10-13  
ASCII character code A-1 to A-2  
Assembler  
absolute expressions 3-24, 4-9  
addressing modes 3-1  
asmb\_sym assembler symbol files 4-4, 9-5,  
9-6  
assembly time relative addressing 2-5  
attributes  
common vs. noncommon 4-2  
section alignment 4-3  
section type 4-4  
short vs. long 4-2  
character set 2-6  
constants 2-11 to 2-15  
character 2-13 to 2-15

floating-point 2-12 to 2-13  
integer 2-11 to 2-12  
cross-reference table format 8-3  
directives 5-1 to 5-74, 6-5 to 6-11, 7-5 to 7-13  
error messages B-1 to B-14  
floating-point 3-6  
HP 64000 files 4-4  
HP-OMF files 9-5  
instructions 3-1  
introduction 1-1, 2-1  
link\_sym linker symbol files 4-4, 9-5, 9-6  
listing, sample program 8-4 to 8-6  
macros 6-1 to 6-11  
name demangling G-2  
name mangling G-2  
object module 8-7  
operation 8-1  
overview 1-1  
program counter 2-10  
relative addressing 3-24 to 3-33  
relocatable expressions 3-24, 4-8 to 4-9  
section attributes  
    assigning 4-6  
section types 4-5  
statement 2-1 to 2-4  
    field 2-1 to 2-2  
    comment 2-2  
    label 2-1  
    operand 2-2  
    operation 2-2  
type 2-2 to 2-4  
    comment 2-4  
    directive 2-3  
    instruction 2-2  
    macro 2-3  
structure control directives 7-5 to 7-13  
symbolic addressing 2-4 to 2-6  
symbols 2-7 to 2-10  
    invalid 2-8  
    relocatable versus absolute 4-8  
    reserved 2-8 to 2-9  
    valid 2-8  
syntax 2-6 to 2-17  
terminator record F-4  
Assembler directives  
    (see Directives, assembler)  
Assembler relocation flags 8-2

**B**

Base address 9-7, 10-14  
BASE linker command 10-14 to 10-15  
Branch instruction labels  
    local (?) 7-4  
BREAK assembler directive 7-6

**C**

CASE linker command 10-16 to 10-17  
Character constants 2-13 to 2-15  
Character set  
    assembler 2-6  
CHIP assembler directive 5-5 to 5-7  
CHIP linker command 10-18 to 10-20  
CLEAR librarian command 13-6  
COMLINE assembler directive 5-8  
Command position dependencies, linker 10-3  
Commands  
    librarian  
        (See Librarian commands)  
    linker  
        (See Linker commands)  
Comment indicators  
    librarian 13-1  
    linker 10-21  
Comment linker command 10-21  
Comment statement 2-4  
COMMON assembler directive 5-9 to 5-10  
COMMON linker command 10-22 to 10-23  
Complex expression 4-9  
Complex relocatable expression 4-9  
Constants  
    (see Assembler constants)  
Continuation character (#) 9-11, 10-49, 10-59  
Conventions, notational viii  
CPAGE linker command 10-24 to 10-25

- 
- CREATE librarian command 13-7  
Cross-reference option 8-3
- D**
- Data record  
    S1 format F-2
- DC assembler directive 5-11 to 5-13
- DCB assembler directive 5-14
- DEBUG\_SYMBOLS command 10-26
- DELETE librarian command 13-8
- Demangling G-1 to G-4
- Directive statement 2-3
- Directives, assembler
- ALIGN 5-4
  - CHIP 5-5 to 5-7
  - COMLINE 5-8
  - COMMON 5-9 to 5-10
  - DC 5-11 to 5-13
  - DCB 5-14
  - DS 5-15
  - ELSEC 5-16
  - END 5-17 to 5-18
  - ENDC 5-19
  - ENDR 5-20
  - EQU 5-21
  - FAIL 5-22 to 5-23
  - FEQU 5-24 to 5-25
  - FOPT 5-26
  - FORMAT 5-27
  - IDNT 5-28
  - IFC 5-29 to 5-30
  - IFDEF 5-31
  - IFEQ 5-32 to 5-33
  - IFGE 5-34
  - IFGT 5-35
  - IFLE 5-36
  - IFLT 5-37
  - IFNC 5-38 to 5-39
  - IFNDEF 5-40
  - IFNE 5-41
  - INCLUDE 5-42
  - IRP 5-43 to 5-44
  - IRPC 5-45 to 5-46
- LIST 5-47  
LLEN 5-48  
macros
- ENDM 6-6
  - LOCAL 6-7 to 6-8
  - MACRO 6-9 to 6-10
  - MEXIT 6-11
- MASK2 5-49  
NAME 5-50  
NOFORMAT 5-27  
NOLIST 5-47  
NOOBJ 5-51  
NOPAGE 5-61  
OFFSET 5-52 to 5-53  
OPT 5-54 to 5-59  
ORG 5-60  
PAGE 5-61  
PLEN 5-62  
REG 5-63  
REPT 5-64  
RESTORE 5-65  
SAVE 5-66  
SECT 5-67 to 5-68  
SECTION 5-67 to 5-68  
SET 5-69  
SPC 5-70  
structured control
- BREAK 7-6
  - FOR...ENDF 7-7 to 7-8
  - IF...THEN...ELSE...ENDI 7-9  
        to 7-10
  - NEXT 7-11
  - REPEAT...UNTIL 7-12
  - WHILE...ENDW 7-13
- TTL 5-71  
XCOM 5-72  
XDEF 5-73  
XREF 5-74
- DIRECTORY librarian command 13-9  
DS assembler directive 5-15
- E**
- EBCDIC character code A-1 to A-2

Effective address syntax 3-14  
**ELSEC** assembler directive 5-16  
**END** assembler directive 5-17 to 5-18  
**END** librarian command 13-10  
**END** linker command 10-27  
**ENDC** assembler directive 5-19  
**ENDM** assembler directive 6-6  
**ENDR** assembler directive 5-20  
**EQU** assembler directive 5-21  
**ERROR** linker command 10-28  
**Error messages**  
  assembler B-1 to B-14  
  librarian D-1 to D-6  
  linker C-1 to C-11  
exists (==) 6-3  
**EXIT** linker command 10-29  
**Expressions** 2-15 to 2-17  
  absolute 3-24, 4-9  
  complex 4-9  
  relocatable 3-24, 4-8 to 4-9  
    complex 4-9  
    simple 4-9  
**EXTERN** linker command 10-30  
**External symbols** 4-7  
**EXTRACT** librarian command 13-11

**F**

**FAIL** assembler directive 5-22 to 5-23  
**Features**  
  assembler 1-1 to 1-2  
  librarian 12-1  
  linker 9-2  
**FEQU** assembler directive 5-24 to 5-25  
**Floating-point**  
  addressing modes 3-14  
  constants 2-12 to 2-13  
  coprocessor 3-6  
  double-precision 5-12  
  single-precision 5-12  
**FOPT** assembler directive 5-26  
**FOR ... ENDF** loop 7-7 to 7-8  
**Formal parameter** 6-2  
**FORMAT** assembler directive 5-27

**FORMAT** linker command 10-31  
**Formats**  
  data record (S1) F-2  
  header record (S0) F-2  
  Motorola S-record F-1 to F-7  
  record count record (S5) F-4  
  symbol record F-1  
**FULLDIR** librarian command 13-12 to 13-13

**G**

**Glossary** E-1 to E-2

**H**

**Header record (S0)** F-2  
**HELP** librarian command 13-14

**I**

**IDNT** assembler directive 5-28  
**IF ... THEN ... ELSE ... ENDI** assembler  
  directive 7-9 to 7-10  
**IFC** assembler directive 5-29 to 5-30  
**IFDEF** assembler directive 5-31  
**IFEQ** assembler directive 5-32 to 5-33  
**IFGE** assembler directive 5-34  
**IFGT** assembler directive 5-35  
**IFLE** assembler directive 5-36  
**IFLT** assembler directive 5-37  
**IFNC** assembler directive 5-38 to 5-39  
**IFNDEF** assembler directive 5-40  
**IFNE** assembler directive 5-41  
**INCLUDE** assembler directive 5-42  
**INCLUDE** linker command 10-32  
**Incremental linking** 9-1, 9-9 to 9-10  
**INDEX** linker command 3-25, 3-27, 10-33 to  
  10-34  
**INITDATA** linker command 10-35  
**INITDATA** section 10-35  
**Instruction operands** 3-3  
**Instruction statement** 2-2  
**Instruction types, variants** 3-2  
**Integer constants** 2-11 to 2-12  
**Introduction**

- assembler 1-1, 2-1
- librarian 12-1
- linker 9-1
- Invocation methods, librarian
  - command file 12-3
  - command line 12-3
  - interactive 12-3
- IRP assembler directive 5-43 to 5-44
- IRPC assembler directive 5-45 to 5-46
- L**
- Librarian
  - command characters
    - asterisk 13-1
    - blanks 13-2
    - comma 13-1
    - parentheses 13-1
    - plus 13-2
    - semicolon 13-1
  - command file comments 13-2
  - commands 13-1
    - ADDLIB 13-4
    - ADDMOD 13-5
    - CLEAR 13-6
    - CREATE 13-7
    - DELETE 13-8
    - DIRECTORY 13-9
    - END 13-10
    - EXTRACT 13-11
    - FULLDIR 13-12 to 13-13
    - HELP 13-14
    - OPEN 13-15
    - QUIT 13-10
    - REPLACE 13-16
    - SAVE 13-17
  - demangled name G-4
  - error messages D-1 to D-6
  - features 12-1
  - function 12-1 to 12-6
  - introduction 12-1
  - invocation methods 12-3
  - listing of commands 13-14
  - listings, sample 14-1 to 14-6
- mangled name G-4
- message severity levels D-1
- overview 1-1
- return codes 12-6
- special characters 13-1
- syntax 13-1 to 13-2
- use of special characters 13-1
- Linker
  - absolute output file 11-9
  - command file 11-4
  - command position dependencies 10-3
  - commands
    - ABSOLUTE 10-7 to 10-8
    - ALIAS 10-9 to 10-10
    - ALIGN 10-11 to 10-12
    - ALIGNMOD 10-13
    - BASE 10-14 to 10-15
    - CASE 10-16 to 10-17
    - CHIP 10-18 to 10-20
    - Comment 10-21
    - COMMON 10-22 to 10-23
    - CPAGE 10-24 to 10-25
    - DEBUG\_SYMBOLS 10-26
    - END 10-27
    - ERROR 10-28
    - EXIT 10-29
    - EXTERN 10-30
    - FORMAT 10-31
    - INCLUDE 10-32
    - INDEX 3-25, 3-27, 10-33 to 10-34
    - INITDATA 10-35
    - LISTABS 10-37
    - LISTMAP 10-38 to 10-39
    - LOAD 10-40 to 10-41
    - LOAD\_SYMBOLS 10-42
    - LOWERCASE 10-43 to 10-44
    - MERGE 10-45 to 10-46
    - NAME 10-47
    - NODEBUG\_SYMBOLS 10-26
    - NOERROR 10-28
    - NOPAGE 10-50 to 10-51
    - ORDER 10-48 to 10-49
    - PAGE 10-50 to 10-51

PUBLIC 10-52 to 10-53  
 RESADD 10-54  
 RESMEM 10-55  
 SECT 10-56  
 SECTSIZE 10-57  
 SORDER 10-58 to 10-59  
 START 10-60  
 SYMTRAN 10-61 to 10-62  
 UPPERCASE 10-63  
 WARN 10-64  
 comment indicator 10-21  
 completion status message 11-3  
 cross-reference table 11-3  
 data record  
     S1 format F-2  
 error messages C-1 to C-11  
 features 9-2  
 header record F-2  
 incremental linking 9-1, 9-9 to 9-10  
 listing, sample 11-1 to 11-18  
 map file 11-5 to 11-8  
 memory space assignment 9-6  
 message severity levels C-1  
 module record F-1  
 operation 11-1  
 overview 1-1, 9-1  
 public symbol table 11-3  
 record count record F-4  
 relocation types 9-8 to 9-9  
 sections 9-2 to 9-6  
     absolute 9-3  
     common 9-4  
     long 9-5  
     noncommon 9-4  
     relocatable 9-3  
     section alignment 9-6  
     short 9-4  
     type 9-5 to 9-6  
 start address 11-3  
 symbol name size G-3  
 symbol record F-1  
 syntax 10-1 to 10-4  
 unresolved externals 11-2

Linking 4-7  
 LIST assembler directive 5-47  
 LISTABS linker command 10-37  
 LISTMAP linker command 10-38 to 10-39  
 LLEN assembler directive 5-48  
 Load address 9-7  
 LOAD linker command 10-40 to 10-41  
 LOAD\_SYMBOLS linker command 10-42  
 LOCAL assembler directive 6-7 to 6-8  
 Local labels (?) 7-4  
 Loop controls 7-4  
 LOWERCASE linker command 10-43 to  
     10-44

**M**

MACRO assembler directive 6-9 to 6-10  
 Macro body 6-9  
 MACRO directive 6-1  
 Macro directives  
     (see Directives, assembler)  
 Macro statement 2-3  
 Macro terminator 6-6  
 Macros 6-1 to 6-11  
     body 6-1  
     call 6-2 to 6-4  
     formal parameter 6-2  
     heading 6-1  
     number of parameters  
         NARG 6-4  
     parameter delimiter (<>) 6-3  
     parameter exists (==) 6-3  
 Mangling G-1 to G-4  
 Manual, description v  
 MASK2 assembler directive 5-49  
 Memory space assignment 9-6 to 9-8  
 MERGE linker command 10-45 to 10-46  
 Message severity levels  
     librarian D-1  
     linker C-1  
 MEXIT assembler directive 6-11  
 Module record F-1  
 Motorola S-record  
     (see S-record format)

## N

NAME assembler directive 5-50  
Name demangling G-1 to G-4  
NAME linker command 10-47  
Name mangling G-1 to G-4  
NARG reserved symbol 2-9, 6-4, 8-3  
NEXT assembler directive 7-11  
NODEBUG\_SYMBOLS linker command 10-26  
NOERROR command 10-28  
NOFORMAT assembler directive 5-27  
NOLIST assembler directive 5-47  
NOOBJ assembler directive 5-51  
NOPAGE assembler directive 5-61  
NOPAGE linker command 10-50 to 10-51  
Notational conventions viii

## O

O opcode error 7-14  
OFFSET assembler directive 5-52 to 5-53  
opcode error (O) 7-14  
OPEN librarian command 13-15  
Operand syntax 3-17 to 3-20  
Operands  
    instruction 3-3  
    syntax 3-17 to 3-20  
OPT assembler directive 5-54 to 5-59  
ORDER linker command 10-48 to 10-49  
ORG assembler directive 5-60

## P

PAGE assembler directive 5-61  
PAGE linker command 10-50 to 10-51  
PLEN assembler directive 5-62  
Program counter 2-10  
Program identifiers 9-10 to 9-11  
Program sections 4-1 to 4-4, 9-2 to 9-5  
Pseudo-Ops  
    (see Directives)  
PUBLIC linker command 10-52 to 10-53

## Q

QUIT librarian command 13-10

## R

Reader's response sheet viii  
Record count record (S5) F-4  
REG assembler directive 5-63  
Registers 3-3 to 3-6  
Relative addressing 3-24 to 3-33  
Relocatable expressions 3-24, 4-8 to 4-9  
Relocatable sections 9-3  
Relocatable symbols 2-9, 4-8  
Relocation flags, assembler 8-2  
Relocation types 9-8 to 9-9  
REPEAT . . . UNTIL assembler directive 7-12  
REPLACE librarian command 13-16  
REPT assembler directive 5-64  
RESADD linker command 10-54  
Reserved symbols 2-8 to 2-9  
    NARG 2-9, 6-4, 8-3  
RESMEM linker command 10-55  
Response forms viii  
RESTORE assembler directive 5-65

## S

SAVE assembler directive 5-66  
SAVE librarian command 13-17  
SECT assembler directive 5-67 to 5-68  
SECT linker command 10-56  
Section alignment 9-6  
SECTION assembler directive 5-67 to 5-68  
Sections  
    absolute 3-21, 9-3  
    common 9-4  
    long 9-5  
    noncommon 9-4  
    program 4-1 to 4-4, 9-2 to 9-5  
    relocatable 3-22, 9-3  
    short 9-4  
    type 9-5 to 9-6  
SECTSIZELinker command 10-57  
SET assembler directive 5-69

- Simple relocatable expression 4-9  
.SIZEOF. operator 2-19  
Software performance report ix  
SORDER linker command 10-58 to 10-59  
SPC assembler directive 5-70  
Special characters  
    linker 13-1  
    linker 10-1  
S-record format F-1 to F-7  
    data record (S1) F-2  
    header record (S0) F-2  
    module record F-1  
    record count record (S5) F-4  
    symbol record F-1  
START linker command 10-60, 10-63  
Starting address 9-7  
    linker commands  
        START 10-63  
.STARTOF. operator 2-18  
Statement  
    assembler 2-1 to 2-4  
Structured control directive code (+) 7-14  
Structured control directives  
    (see Directives, assembler)  
Structured control expressions 7-1 to 7-3  
Structured directives, nesting 7-14  
Subsection 9-3  
Symbol  
    name size  
        linker G-3  
Symbol record F-1  
Symbolic addressing 2-4 to 2-6  
Symbols 2-7 to 2-10  
    absolute 2-9, 4-8  
    external 4-7  
    invalid 2-8  
    relocatable 2-9, 4-8  
    reserved  
        NARG 2-9  
    valid 2-8  
SYMTRAN linker command 10-61 to 10-62  
Syntax  
    addressing mode 3-15 to 3-20
- assembler 2-6 to 2-17  
effective address fields 3-14  
librarian 13-1 to 13-2  
linker 10-1 to 10-4
- T**
- TTL assembler directive 5-71
- U**
- U undefined label error 7-14  
Undefined label error (U) 7-14  
UPPERCASE linker command 10-63
- W**
- WARN linker command 10-64  
WHILE . . . ENDW assembler directive 7-13
- X**
- XCOM assembler directive 5-72  
XDEF assembler directive 5-73  
XREF assembler directive 5-74

## Reader's Response

Please complete and return the following Reader's Response form to help us improve our documentation. Your comments and suggestions are always appreciated.

Product Name \_\_\_\_\_

Manual Date/Version \_\_\_\_\_

Your Name \_\_\_\_\_

Title \_\_\_\_\_

Company \_\_\_\_\_

Address \_\_\_\_\_

Phone Number \_\_\_\_\_

What is your level of programming experience?

In this language?

Advanced

Advanced

Intermediate

Intermediate

Beginner

Beginner

Circle the number which best expresses your rating of the information in this manual (*5 is the highest rating*):

Complete?

1

2

3

4

5

Easy to understand?

1

2

3

4

5

Accurate?

1

2

3

4

5

Helpful examples?

1

2

3

4

5

Well organized?

1

2

3

4

5

Helpful procedures?

1

2

3

4

5

Easy to Find?

1

2

3

4

5

At an appropriate level?

1

2

3

4

5

Easy to Read?

1

2

3

4

5

Overall rating?

1

2

3

4

5

3.) Which programming language do you use?

---

4.) Does the manual provide all the necessary information? If not, what is missing?

---

5.) Are more examples needed? If so, where?

---

6.) Did you have any difficulty understanding descriptions or wording? If so, where?

---

7.) Which sections of this manual are the most important? The most helpful?

---

8.) What are the best features of this manual?

---

9.) What are the areas in this manual that need improvement?

---

10.) Did you find any errors in this manual? (Specify section and page number.)

---



## **Software Performance Report Instructions**

If you encounter problems or errors when using Microtec Research software products (including documentation), you can send a Software Performance Report (SPR):

- Electronically through e-mail to [support@mri.com](mailto:support@mri.com)

or

- By completing the SPR form on the next page and returning it to:

**Microtec Research  
Software Performance Reports  
2350 Mission College Boulevard  
Santa Clara, CA 95054**

All SPRs are directed to our Technical Support Department for analysis and response.

To make it easier for us to respond quickly to your problem, please include the following:

1. Give a complete description of the problem, error, or suggestion, including the exact wording of any error messages.
2. If possible, isolate the problem by providing a small example.
3. If you are mailing or FAXing an SPR and your error example is longer than one page of source code, please provide all information on a tape or floppy disk.
4. Include command files, listings, load maps, include files, data files, and all other relevant material with the message.
5. If applicable, send sample input and output listings.
6. If the output is from a compiler, send a mixed (source/assembly code) listing.
7. Be sure to include the following information:
  - Product name and version number
  - Serial number
  - Name of contact
  - Company purchased from
  - Company name and a shipping address
  - Phone number and/or FAX number
  - Host machine
  - Host operating system and version number
  - Customer impact

All communications will be acknowledged the next working day.



## Software Performance Report

Product:	Customer:
Version:	Company Name/Address:
Serial Number:	
Purchased From:	Phone Number:
	E-Mail Address:

Host:       Apollo       PC  
               DECstation       Sun-3  
               HP 9000 Series 300/400       Sun-4/SPARC  
               HP 9000 Series 700       VAX/VMS  
               IBM RS/6000       Other \_\_\_\_\_

Operating System \_\_\_\_\_ Version \_\_\_\_\_

Report Type: Customer Impact:

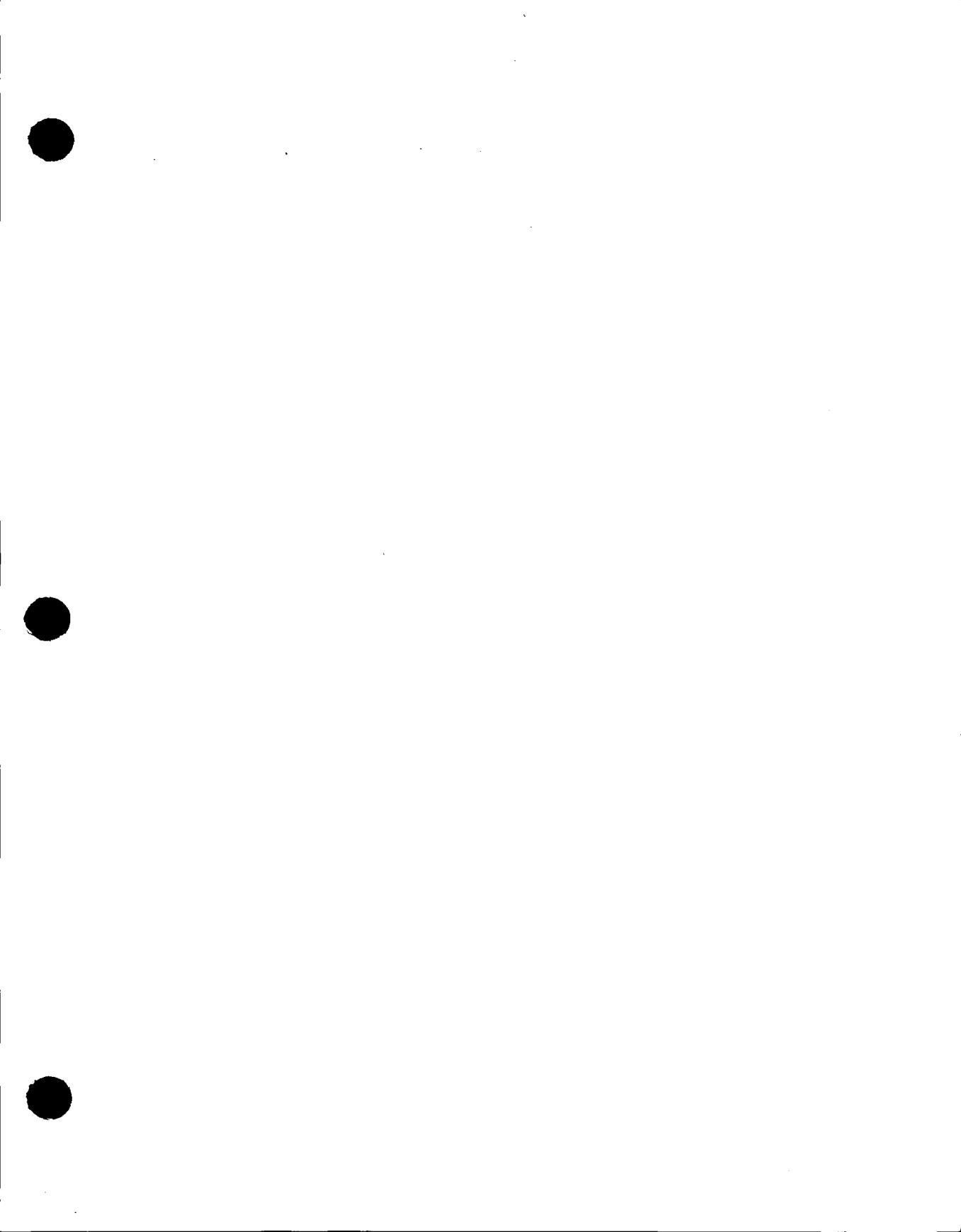
(Attach additional pages if necessary to complete the items below.)

Problem Description:

Commands or lines of code used to duplicate problem:

Workaround:







***Microtec  
Research***

2350 Mission College Blvd.

Santa Clara, CA 95054

Tel. 408.980.1300

Toll Free 800.950.5554

FAX 408.982.8266



*Microtec*  
*Research*

**ASM68K**  
*Reference  
Manual*

100009-008

## TRADEMARKS

Microtec® and Paragon® are registered trademarks of Microtec Research, Inc.  
MRI ASMTM, MRI CTM, MRI FORTRANTM, MRI Pascal™, MRI XRAY™, Source Explorer™,  
XHM302™, XRAY™, XRAY Debugger™, XRAY In-Circuit Debugger™, XRAY In-Circuit  
Debugger Monitor™, XRAY MasterWorks™, XRAY/MTD™, XRAY180™, XRAY29K™,  
XRAY51™, XRAY68K™, XRAY80™, XRAY86™, XRAY88K™, XRAY960™, XRAYG32™,  
XRAYH83™, XRAYH85™, XRAYM77™, XRAYSP™, XRAYTX™, and XRAYZ80™ are  
trademarks of Microtec Research, Inc.

Other product names mentioned in this document are trademarks or registered trademarks  
of their respective companies.

## RESTRICTED RIGHTS LEGEND

If this product is acquired under the terms of a: *DoD contract*: Use, duplication, or disclosure by  
the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of 252.227-7013.  
*Civilian agency contract*: Use, reproduction, or disclosure is subject to 52.227-19 (a) through (d)  
and restrictions set forth in the accompanying end user agreement. Unpublished rights reserved  
under the copyright laws of the United States.

MICROTEC RESEARCH, INCORPORATED  
2350 MISSION COLLEGE BLVD.  
SANTA CLARA, CA 95054

© Copyright 1988, 1989, 1990, 1991, 1992 Microtec Research, Inc. All rights reserved. No part of  
this publication may be reproduced, transmitted, or translated, in any form or by any means,  
electronic, mechanical, manual, optical or otherwise, without prior written permission of Microtec  
Research, Inc.

REV.	REVISION HISTORY	DATE	APPD.
-002	Updated from V 6.3 to V 6.4 the VAX/VMS and UNIX chapters.	6/88	L.C.
-003	Updated from V 6.4 to V 6.5 the VAX/VMS and UNIX chapters.  Added the IBM-PC/MS-DOS chapter.	8/89	L.C.
-004	Printed at 80% for new packaging.	11/89	L.C.
-005	For V 6.4, merged the IBM-PC/MS-DOS chapter into this guide.  Printed V 6.4 at 80% for new packaging.	12/89	L.C.
-006	Updated from V 6.5 to V 6.6 the VAX/VMS, UNIX, and IBM-PC/MS-DOS chapters.  Added the LLEN and P=68332 assembler command line flags.	2/90	L.C.
-007	Updated to Version 6.8A.	9/91	L.L.
-008	Updated to Version 6.9.	8/92	D.C.



## About This Guide

The Microtec Research *ASM68K User's Guide* describes how to use the Microtec Research ASM68K assembler package software. It gives detailed descriptions of the various ways to invoke the components of the package, assembler, linker, and object module librarian. Additionally, the manual describes command line options that let you control output listing and object module generation for the programs.

This guide contains the following:

- Chapter 1, *Introduction*, provides high-level descriptions of the assembler, linker, object module librarian, and the optional XRAY Debugger.
- Chapter 2, *UNIX / DOS User's Guide*, describes how to invoke the assembler, linker, and object module librarian on host computers using a UNIX or UNIX-like operating system, PC-DOS, or MS-DOS. The chapters lists and describes the operating-system specific command syntax and command line options for each tool.
- Chapter 3, *VAX / VMS User's Guide*, describes how to invoke the assembler, linker and object module librarian on VAX computers running VMS. The chapters lists and describes the operating-system specific command syntax and command line options for each tool.
- Appendix A, *HP 64000 Development System Support*, describes how the ASM68K assembler package interacts with the HP 64000 Development System.

Once the programs have been installed, this *User's Guide* can be used in conjunction with the *Reference Manual*.

## About the Documentation Set

The documentation set describing the ASM68K Assembler consists of the following publications in addition to this *User's Guide*:

- The *Microtec Research Reference Manual* describes the assembler syntax and directives, linker commands, and librarian commands. Sample program listings are used as examples for the assembler, linker, and object module librarian. Error and warning messages are described.

- The *Microtec Research Installation Guide* describes how to install the ASM68K package for those host computers which do not support Flexible Licensing.

These documents assume a working knowledge of the Motorola 68000 microprocessor. For background information, see the list of references in the next section.

## Related Publications

The *Microtec Research ASM68K User's Guide* is written for the experienced program developer and assumes the developer has a working knowledge of the Motorola 68000 family of microprocessors. Although it provides several useful and informative program examples, this documentation does not describe the microprocessor itself. For such information, refer to:

- *Motorola Programmer's Reference Manual*, M68000PM/AD REV1
- *The Motorola M68000 Resident Structured Assembler Reference Manual*, M68KMASM/D8
- *Motorola 68000 Microprocessor User's Manual*, M68000UM(AD4)
- *Motorola 68020 Microprocessor User's Manual*, MC68020UM/AD REV 1
- *Motorola MC68881/MC68882 Floating-Point Coprocessor User's Manual*, MC68881UM/AD REV 1
- *Motorola 68030 Enhanced 32-Bit Microprocessor User's Manual*, MC68030UM/AD
- *MC68040 Enhanced 32-Bit Third Generation Microprocessor User's Manual*, MC68040UM/AD
- *Motorola 68851 Paged Memory Management Unit User's Manual*, MC68851UM/AD
- *Motorola CPU Central Processor Unit Reference Manual*, CPU32RM/AD
- *Motorola M68000 Family Reference*, M68000FR/AD
- *Motorola M68040 Microprocessor User's Manual*, M68040UM/AD

This manual assumes that you are familiar with programming and with the C programming language. For general information about C, refer to the following publications:

- *C: A Reference Manual*, 3rd ed., by Samuel P. Harbison and Guy L. Steele Jr., Prentice-Hall, Inc., 1991.

- *The C Programming Language*, 2nd ed., Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1988.

Refer to the following publications for further information about the software development process and the C++ programming language:

- *The Annotated C++ Reference Manual*, by Margaret Ellis and Bjarne Stroustrup, AT&T Bell Laboratories, 1990.
- *C++ for C Programmers*, by Ira Prohl, Addison-Wesley, 1989.
- *C++ Primer*, by Stanley Lippman, Addison-Wesley, 1989.
- *C++ Programming Language*, by Bjarne Stroustrup, AT&T Bell Laboratories, 1987.
- *Programming in C++*, by Steve Dewhurst and Kathy Stark, Prentice-Hall, 1989.

For information about other Microtec Research 68000 toolkit components, refer to the following publications:

- *MCC68K Documentation Set*, Microtec Research, Inc.  
This documentation set describes how to use the MCC68K ANSI C Cross Compiler.
- *XRAY68K Documentation Set*, Microtec Research, Inc.  
This documentation set describes how to use the XRAY68K Debugger.
- *CCC68K Documentation Set*, Microtec Research, Inc.  
This documentation set describes how to use the CCC68K C++ Compiler.
- *C++ Capsule Class Library*, Microtec Research, Inc.  
This documentation set describes how to use the reusable data structure classes provided with the C++ Capsule Class Library.
- *XDM68K XRAY In-Circuit Debugger Monitor Documentation Set*, Microtec Research, Inc.  
This documentation set describes how to use the XDM68K In-Circuit Debugger Monitor with XRAY.
- *Microtec Research Flexible License Manager Documentation Set*, Microtec Research, Inc.  
This documentation set describes how to install and manage licenses for Microtec Research products that use Flexible Licensing.

## Notational Conventions

Examples in this guide are identified by operating system name. The name UNIX refers to any UNIX or UNIX-like operating system, such as SunOS and HP-UX. The name DOS refers to both MS-DOS and PC-DOS systems.

This guide uses the notational conventions shown in Table P-1 (unless otherwise noted).

Table P-1. Notational Conventions

Symbol	Name	Usage
{ }	Curly Braces	Encloses a list from which you must choose an item.
[ ]	Square Brackets	Encloses items that are optional.
...	Ellipsis	Indicates that you may repeat the preceding item zero or more times.
	Vertical Bar	Separates items in a list.
	Punctuation	Punctuation other than that described here must be entered as shown.
	Typewriter font	Represents user input in interactive examples.
	<i>Italics</i>	Indicates a descriptive item that should be replaced with an actual item.

## Microprocessor References

The 68000 family of microprocessors includes the 68000, 68008, 68010, 68012, 68020, 68030, 68302, CPU32, 68040, 68881, 68882, 68851, 68HC000, 68HC001, 68EC000, 68EC020, 68EC030, 68EC040 and the CPU32 family: 68330, 68331, 68332, 68333, 68340. In this manual, they are referred to collectively as 68000.

## Questions and Suggestions

To help us respond to questions or suggestions regarding this product, we have provided two response forms in the back of this documentation.

The first form is a Reader's Response sheet, which is used to help us correct and improve our documentation. We always appreciate your comments and, by com-

pleting this form, you can participate directly in the revisions of our publications. The Reader's Response sheet is directed to Technical Publications.

The second form is a Software Performance Report which should be completed if you encounter any errors or problems with Microtec Research software. This report is directed to Technical Support.



# Contents

---

## Preface

About This Guide .....	v
About the Documentation Set .....	v
Related Publications .....	vi
Notational Conventions .....	viii
Microprocessor References .....	viii
Questions and Suggestions .....	viii

## 1 Introduction

Components of the Microtec Research ASM68K Package .....	1-1
Assembler .....	1-1
Linker .....	1-1
Object Module Librarian .....	1-2
Other Microtec Research Products .....	1-2
C Compiler .....	1-2
XRAY Debugger .....	1-2
Data Flow .....	1-3

## 2 UNIX / DOS User's Guide

Introduction .....	2-1
ASM68K Assembler .....	2-1
LNK68K Linker .....	2-1
LIB68K Librarian .....	2-2
ASM68K Assembler .....	2-2
Invocation Syntax .....	2-2
File Name Defaults .....	2-4
Environment Variables .....	2-5
Placement of Temporary Files (DOS only) .....	2-5
Command Line Flags .....	2-5
Invocation Examples .....	2-12
LNK68K Linker .....	2-13
Invocation Syntax .....	2-13
Environment Variables .....	2-16
MRI_68K_LIB .....	2-16
TMP (DOS only) .....	2-17
File Name Defaults .....	2-17
Continuation Character .....	2-18
Invocation Examples .....	2-18
Invoking the Linker with a Command File .....	2-18

---

Invoking the Linker Without a Command File .....	2-19
LIB68K Librarian .....	2-21
Invocation Syntax .....	2-21
File Name Defaults .....	2-22
Invocation Examples .....	2-23
Invoking the Librarian Interactively .....	2-23
Invoking the Librarian from the Command Line .....	2-23
Invoking the Librarian with a Command File .....	2-23
Utility Programs .....	2-24
IEE2AOUT Conversion Program .....	2-24
Section Typing .....	2-25
Relocation File Conversion .....	2-25
Absolute File Conversion .....	2-26
Creating Your own Library .....	2-26
Return Codes .....	2-28

### 3 VAX / VMS User's Guide

Introduction .....	3-1
ASM68K Assembler .....	3-1
LNK68K Linker .....	3-1
LIB68K Librarian .....	3-2
ASM68K Assembler .....	3-2
Invocation Syntax .....	3-2
File Name Defaults .....	3-4
Command Line Flags .....	3-5
Invocation Examples .....	3-12
LNK68K Linker .....	3-12
Invocation Syntax .....	3-13
File Name Defaults .....	3-16
Invocation Examples .....	3-16
Invoking the Linker with a Command File .....	3-16
Invoking the Linker from the Command Line .....	3-17
Continuation Character .....	3-18
LIB68K Librarian .....	3-18
Invocation Syntax .....	3-18
File Name Defaults .....	3-20
Invocation Examples .....	3-20
Invoking the Librarian Interactively .....	3-20
Invoking the Librarian from the Command Line .....	3-20
Invoking the Librarian with a Command File .....	3-21
Utility Programs .....	3-21
IEE2AOUT Conversion Utility .....	3-21
Section Typing .....	3-23
Relocation File Conversion .....	3-23

## Contents

---

Absolute File Conversion .....	3-24
Creating Your own Library .....	3-24
Creation of Temporary Files .....	3-26
VMS Return Codes .....	3-27

## Appendix A: HP 64000 Development System Support

Overview .....	A-1
HP 64000 Considerations .....	A-1

## Figures

---

Figure 1-1. Components of the Microtec Research ASM68K Package .....	1-4
--	-----

## Tables

---

Table P-1. Notational Conventions .....	viii
Table 2-1. UNIX/DOS Assembler File Name Extensions .....	2-4
Table 2-2. UNIX/DOS Assembler Command Line Flags .....	2-6
Table 2-3. UNIX/DOS Linker File Name Extensions .....	2-17
Table 2-4. UNIX/DOS Librarian File Name Extensions .....	2-22
Table 2-5. IEEE Section Names .....	2-25
Table 3-1. VMS Assembler File Name Extensions .....	3-4
Table 3-2. VMS Assembler Command Line Flags .....	3-6
Table 3-3. VMS Linker File Name Extensions .....	3-16
Table 3-4. VMS Librarian File Name Extensions .....	3-20
Table 3-5. IEEE Section Names .....	3-23
Table A-1. Section Types for the HP 64000 Sections .....	A-2
Table A-2. Modifications Required for the 68000 Monitor .....	A-3



## Components of the Microtec Research ASM68K Package

The Microtec Research ASM68K assembler package consists of an assembler, a linker, and an object module librarian. They provide an integrated system for developing software applications for the target Motorola 68000 processor. The components of the Microtec Research ASM68K package are described in the following sections.

### Assembler

The Microtec Research ASM68K Assembler converts assembly language programs to relocatable object code. Object modules are suitable for linking with other modules or with libraries of modules.

The assembler accepts source program statements that are syntactically compatible with those accepted by the 68000 family assemblers. Many additional directives are provided for versatility. The ASM68K Assembler processes macros, conditional assembly statements, and compiler-like structure directives. The assembler generates a cross-reference table as well as a standard symbol table. Generated code and data can be placed in multiple named or numbered sections.

After assembly, the linker links the object modules, which are then suitable for downloading and execution on any 68000 family microprocessor.

### Linker

The Microtec Research LNK68K Linker combines relocatable object modules into a single absolute object module. You can generate object modules in the following formats:

- Motorola S-record
- Hewlett-Packard 64000 format (HP-OMF)
- IEEE-695

The linker also supports the combining of multiple relocatable object modules into a single relocatable module, which subsequently can be relinked with other modules. This feature is referred to as incremental linking.

If one of the input files is a library of object modules, the linker automatically loads only those modules from the library that are referenced by the other named object modules. The linker produces a link map that shows the final location of all modules and sections, and the final absolute values of all symbols. It also generates a cross-reference listing, showing which modules refer to each global symbol.

You can execute the linker in batch mode by using a command file or in a modified batch mode by specifying a command file plus additional object modules/libraries on the command line. The linker reports unresolved external symbols and link errors on the link map or on the terminal.

## Object Module Librarian

The Microtec Research LIB68K Object Module Librarian lets you maintain a collection of relocatable object modules that reside in one file. Libraries let you automatically load frequently-used object modules without concern for the specific names and characteristics of the modules.

By using the librarian, you can format and organize library files that will subsequently be used by the linker. You can add, delete, replace, and extract modules, as well as obtain a directory listing of library contents.

## Other Microtec Research Products

### C Compiler

The Microtec Research C Compiler converts both ANSI C and C source programs into tight, efficient assembly language code for use with the ASM68K Assembler. You can use the compiler to create ROMable programs, generate position-independent code and data, support register-relative data addressing, and produce debugging information for the XRAY Debugger.

### XRAY Debugger

The Microtec Research XRAY Debugger lets you monitor and control the execution of programs at the source level using a window-oriented user interface. You can examine or modify the value of program variables, procedures, and addresses using the same source-level terms, definitions, and structures defined in the original source code. The interactive debugger gives you complete control of the program through an execution environment such as a simulator, in-circuit emulator, single board computer, or target monitor.

XRAY68K's powerful command language allows simple and complex breakpoint setting, single-stepping, and continuous variable monitoring. Input and output can

be directed to/from files, buffers, or viewports. In addition, a sophisticated macro facility allows complex command sequences to be associated with events such as the execution of a specific statement or the accessing of a specific data location. These features let you isolate errors and patch your source code.

## Data Flow

Figure 1-1 shows the components of the Microtec Research ASM68K package and the way in which they communicate with each other and with the host computer system.

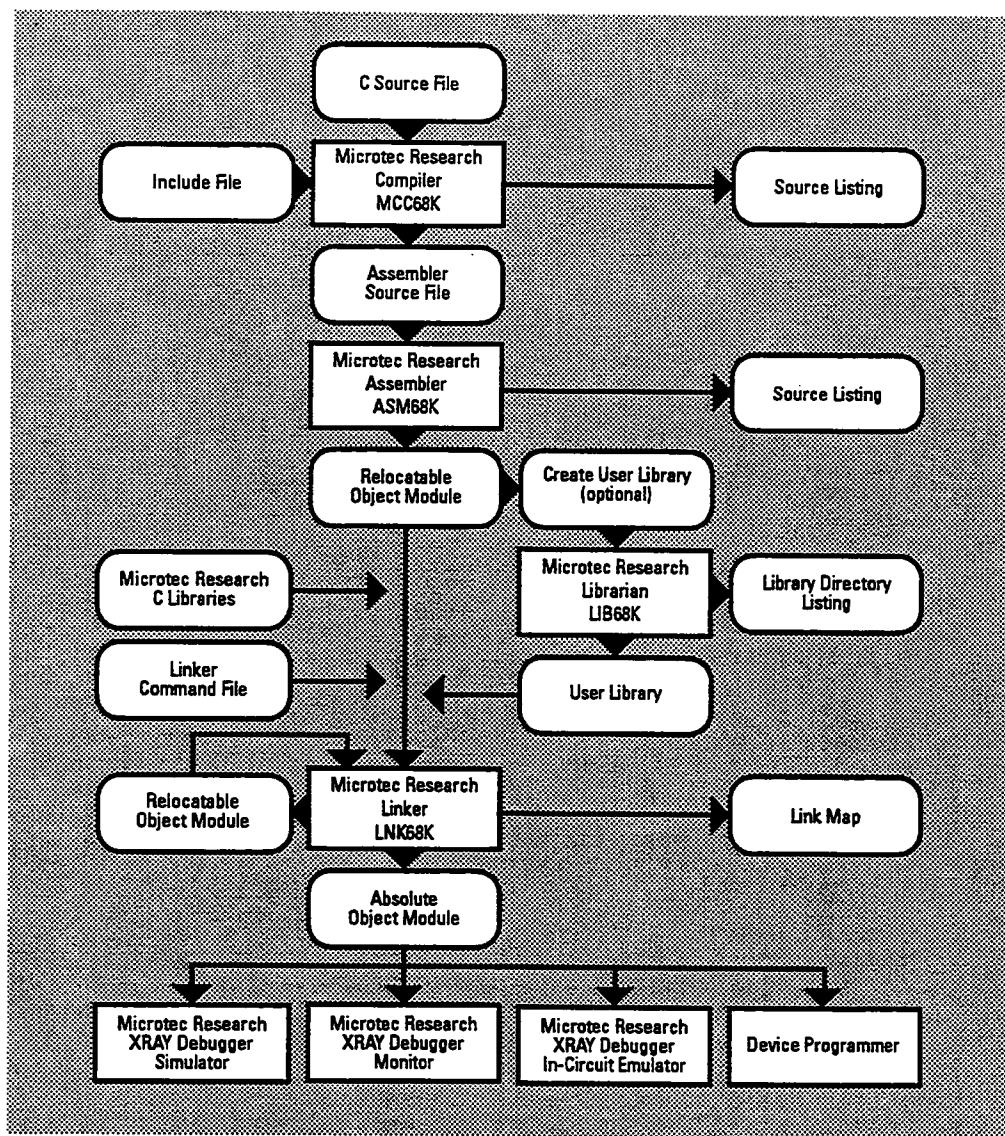


Figure 1-1. Components of the Microtec Research ASM68K Package

## Introduction

This chapter describes the basic use of the assembler, linker, and object module librarian for use on all hosts running UNIX systems and DOS operating systems. The method for executing the programs is identical for these systems, except where noted.

The chapter is divided into sections, one for each tool; each section contains descriptions of the following information:

- Invocation syntax
- Environment variables, if any
- Input and output file name defaults
- Command line flag descriptions, if any
- Invocation examples

The last sections of the chapter describe the operating system return codes generated by the tools and utility programs that come with the assembler package.

### ASM68K Assembler

The Microtec Research ASM68K Assembler converts assembly language programs to relocatable object code. Object modules are produced in IEEE-695 format. After assembly, object modules are combined by the linker.

### LNK68K Linker

The Microtec Research LNK68K Linker allows multiple relocatable object modules to be combined into a single relocatable module that subsequently can be relinked with other modules (i.e., incremental linking). The linker can also combine relocatable object modules into a single absolute object module in one of the following formats:

- Motorola S-record format
- IEEE-695 format
- Hewlett-Packard 64000 format (HP-OMF)

The LNK68K Linker gives you the capability to load your software at locations that match your unique hardware environment. You can specify:

- Where to load sections
- Order to load sections
- Size of sections to use

If one or more of the `linker` input files is a library of object modules, the linker automatically loads all modules from the library that are referenced by the individually-named object modules.

## LIB68K Librarian

The Microtec Research LIB68K Object Module Librarian lets you maintain a collection of relocatable object modules that reside in one file. By using the librarian, you can format and organize library files that will subsequently be used by the linker.

## ASM68K Assembler

The Microtec Research ASM68K Assembler produces object code typically for later use by the LNK68K Linker and LIB68K Librarian.

### Invocation Syntax

The following command line syntax shows how to invoke the ASM68K Assembler. Error messages as a result of erroneous command line entries are written to the standard error output device. This device is `stderr` for UNIX systems and your terminal or a file for a DOS system. The line continuation character, the backslash (\), continues the command line entry on the next line for UNIX systems only.

#### Syntax:

```
% asm68k [-b] [-D sym[=value]] [-f flag_list] [-h] [-H asmb_symfile]  
[-I pathname ...] [-l | -L] [-o object_file] [-V] source_file
```

#### Description:

<code>asm68k</code>	Invokes the assembler.
<code>-b</code>	Stores intermediate information in a temporary file (instead of virtual memory) between pass 1 and 2 of the assembler. This option is valid for UNIX systems only; DOS systems always use a temporary file. (default= <code>-b</code> )
<code>-D <i>sym[=value]</i></code>	Defines the symbol <i>sym</i> at assembly time. The parameter <i>value</i> must be an integer constant. If <i>value</i> is not specified, <i>sym</i> is set to 1.

The value of *sym* will be overridden by the first SET directive applied to *sym* in the source file. No EQU directives can be applied to *sym*.

**-f** *flag\_list*

Enables or disables the internal assembler listing control switches. A list of command line flags for *flag\_list* is shown in Table 2-2.

**-h**

Generates an HP 64000 compatible (HP-OMF) **asmb\_sym** symbol file. The **d** flag is automatically set, placing local symbols in the output object module. The default file name is *source\_file.A*.

For more information on HP 64000 development support, see Appendix A.

**-I** *pathname*

Specifies a path to be searched to locate files that are to be included using the INCLUDE directive. For more information on the INCLUDE directive, see *Assembler Directives* in the *Microtec Research ASM68K Reference Manual*.

The assembler first searches for include files in the current working directory. It next searches the directory specified by *pathname*. A maximum of four paths can be specified. For instance, **-Ipath1 -Ipath2** and so on.

**-l**

**-L**

Produces a listing file which is written to the standard output device. The standard output can be redirected to a file.

**-o** *object\_file*

Overrides the default object module file name with the specified file name. The default object file name is the *source\_file*; the extension listed in Table 2-1 replaces the extension of *source\_file*.

The **-f noobject** option will override the **-o** option, and an output object module will not be produced.

**-v**

Lists the current version number to the standard output device and exits the assembler. All other command line entries are ignored. The version number is always displayed on the output listing.

(default=no version number displayed)

*source\_file*

Specifies the assembler source file. The default file name extensions are listed in Table 2-1. The source file must be the

last entry on the command line, excluding redirected standard output listings. Only one *source\_file* is permitted on a command line.

If you do not specify options, the following defaults will be used:

no listing file

*source\_file.obj* in IEEE-695 format (DOS)

*source\_file.o* in IEEE-695 format (UNIX)

Refer to Table 2-2 for the default flags that will be used.

## File Name Defaults

If *source\_file* does not contain an extension, the assembler assumes the extensions listed in Table 2-1. An output object module is produced by default, but if no output file name is specified on the command line, the assembler uses *source\_file* as the root file name and appends the default file name extensions listed in Table 2-1.

**Table 2-1. UNIX/DOS Assembler File Name Extensions**

File	UNIX	DOS
Assembler source file	.S	.SRC
HP-OMF asmb_sym file	.A	.A
Relocatable object file	.O	.OBJ

File names can include full file specification information including:

Path name

File name

Extension

If you do not supply a path name, file name, or extension, the current defaults are used.

## Environment Variables

This section describes environment variables that affect ASM68K Assembler operation. Environment variables must be set before invoking the assembler.

### Placement of Temporary Files (DOS only)

You can control the location where ASM68K and LNK68K open temporary files by setting the environment variable **TMP**.

#### Example:

```
set TMP=D:\PATH (DOS)
```

This command causes a subsequent invocation of ASM68K or LNK68K to use device D: and directory \PATH for temporary files.

For DOS, the environment variable **TMP** should be set in your **AUTOEXEC.BAT** file. If this command is not specified or an invalid directory is used, the temporary files will be opened in the current directory.

## Command Line Flags

Assembler command flags for *flag\_list* in the **-f flag\_list** option are listed in Table 2-2. With the **-f flag\_list** option, internal assembly control switches are enabled and disabled.

Abbreviations for the flags are also shown. After entering an abbreviation, any number of characters can be specified up to the flag's full spelling. Some flags may be negated by using the prefix **no**, as shown in the table. Multiple items in the list are separated by commas, and enclosed within double quotes. Items requiring the use of parentheses must be enclosed in double quotes. Spaces cannot appear in the list unless the list is enclosed within double quotes. If conflicting flags are specified or a flag is specified multiple times, the last flag specified overrides all previous conflicting flags.

More information about the flags can be obtained in the description of the **OPT** directive in the *Microtec Research ASM68K Reference Manual*. Flags entered on the command line have the effect of inserting an **OPT** directive at the beginning of the source file. However, flags on the command line do not override other **OPT** directives in the source file.

Table 2-2. UNIX/DOS Assembler Command Line Flags

Flags	Meaning (positive form)	Default
abspcadd	When this flag is set, an absolute expression in conjunction with the mnemonic PC (i.e., S(PC)) refers to an absolute address accessed through PC relative mode rather than to a relative displacement added to the current program counter.  For more information, refer to <i>Assembler Syntax for Effective Address Fields</i> in Chapter 3 of your <i>ASM68K Reference Manual</i> .	abspcadd
brb		
brs		
b	Forces forward references in relative branch instructions (Bcc, BRA, BSR) to use the short form of the instruction (8-bit displacement). If nobrl, nobrs, or nob is indicated, the assembler will be set to the default brw.	brw
brl	Forces forward references in relative branch instructions (Bcc, BRA, BSR) to use the long address (32 bits). If OPT OLD is in effect in conjunction with the brl flag, the word displacements are used. If nobrl is indicated, the assembler will be set to the default brw.  This command is effective only on 32-bit chips (68EC020/030/040, 68020/30/40, and CPU32 family).  For 32-bit processors, when OPT OLD has not been specified, 32-bit displacements are used. When OPT OLD has been specified, 16-bit displacements are used. In all other processor modes, 16-bit displacements are used.	brw
brw	Forces 16-bit displacements to always be used in relative branch instructions (Bcc, BRA, BSR) that have forward references. brw cannot be negated.	brw

(cont.)

Table 2-2. UNIX/DOS Assembler Command Line Flags (cont.)

Flags	Meaning (positive form)	Default
case	Retains case-sensitivity of symbols. For example, abc and ABC are treated as different symbols.	case
ca		
cex		cex
c	Lists all lines of object code that are generated by the DC directive. The nocex flag only lists the first line of a DC directive.	
cl		cl
i	Lists instructions that are not assembled due to conditional assembly statements.	
cre	Lists the cross-reference table on the output listing. This option overrides the symbol table output option, t. If both t and cre are used, a cross-reference table is generated.	nocre
d	Places debug and symbol information in the output object module. This flag is automatically set if you generate an asmb_sym file by using the -H or -h option.	nod
e	Lists lines with errors on the standard error output device, as well as on the output listing.	e
frl	Forces instructions that contain a forward reference to an absolute (nonrelocatable) address to use a 32-bit address instead of a 16-bit address. This command is effective only on 32-bit chips (68EC020/030/040, 68020/30/40, and CPU32 family).	32-bit address or frl
frs		
f	Forces instructions that contain a forward reference to an absolute (nonrelocatable) address to use a 16-bit address instead of a 32-bit address.	nofrs
g	Lists assembler-generated symbols in the symbol or cross-reference table. If d is also specified, these symbols are included in the object module as well.	nog

(cont.)

**Table 2-2. UNIX/DOS Assembler Command Line Flags (cont.)**

<b>Flags</b>	<b>Meaning (positive form)</b>	<b>Default</b>
llen=n	Sets the length of the line on the source listing to be <i>n</i> columns long. The <i>n</i> can be a number between 37 and 1120, inclusive.	llen=132
mc	Prints macro calls in the program listing.	mc
md	Prints macro definitions on the program listing.	md
mex	Lists macro and structured control directive expansions on the program listing.	mex
m		
nest=n	Sets <i>n</i> nesting levels for macros. The maximum number of nesting levels allowed is 100 (UNIX) or 20 (DOS).	nest=100 (UNIX) or nest =20 (DOS)
o	Produces the output object module.	o
old	Specifies that the interpretation of the <b>b1</b> flag and the <b>L</b> size qualifiers be 16-bit displacements for Bcc instructions when <b>OPT BRL</b> or <b>OPT -B</b> are specified, or when explicit <b>L</b> qualifiers are used in 68020 mode. This is useful when migrating 68000 programs to 32-bit processors (68EC020/030/040, 68020/30/40, and CPU32 family).	noold
op=n	Resets the maximum number of optimization loops that the assembler will perform if the <b>opnop</b> flag is set. The assembler will discontinue looping if there is a pass in which no optimization occurs.	op=3

(cont.)

Table 2-2. UNIX/DOS Assembler Command Line Flags (cont.)

Flags	Meaning (positive form)	Default
opnop	Removes for optimization purposes any NOPs that have been generated by the assembler because of forward references. The assembler will often assume the worst case in determining the size of forward-referenced variables and will reserve the maximum space (usually 32 bits) for them. During the second pass of the assembler, NOPs will be generated if the operand represented by the variable can be used in an instruction format that is more compact than the size reserved during the first pass.	noopnop
p= <i>type</i> [/ <i>cotype</i> ] . . .	<p>Specifies a set of compatible processors and coprocessors. The <i>type</i> can be any processor of the 68000 family. A list of target processors and instruction sets is shown in Table 5-2 in the <i>Reference Guide</i>.</p> <p>The <i>cotype</i> can be either 68881 or 68851 with the following restrictions for <i>type</i>:</p> <ul style="list-style-type: none"> <li>• If the <i>cotype</i> is 68881, the 68040 processor <i>type</i> is illegal.</li> <li>• If the <i>cotype</i> is 68851, the 68030 and 68040 processor <i>types</i> are illegal.</li> </ul>	68000/68881
pco p	<p><b>Example:</b></p> <p><code>p=68020/68881/68851</code></p> <p>It is an error to specify an incompatible set of processors, such as <code>p=68030/68851</code>.</p> <p>Uses the program counter relative addressing mode on backward references within an absolute (<b>ORG</b>) section whenever possible. This flag is only applied to instructions where the relative addressing mode is legal and the displacement from the program counter fits within the 16-bit field provided.</p>	nopco

(cont.)

Table 2-2. UNIX/DOS Assembler Command Line Flags (cont.)

Flags	Meaning (positive form)	Default								
pcr	<p>Specifies that a PC-relative address mode will be used on references from a relocatable section to the same section. This applies to all instructions for which the PC-relative address mode is legal.</p> <p>The pcr flag differs from pcs in that pcs applies to all relocatable sections within this module and assumes you know the boundaries. The pcr flag only affects intrasection expressions in the current module (expressions for which the assembler has enough information to generate the correct offset).</p>	pcr								
pcs r	Uses the PC-relative address mode on backward references from a relocatable section to the same section or to a different relocatable section. This address mode will be used for all instructions for which the PC-relative address mode is legal.	nopcs								
quick	<p>Changes the MOVE, ADD, and SUB instructions to the more efficient MOVEQ, ADDQ, and SUBQ instructions when possible.</p> <table> <thead> <tr> <th>Before the Change</th> <th>After the Change</th> </tr> </thead> <tbody> <tr> <td>MOVE.L #data, Dn</td> <td>MOVEQ #data, Dn</td> </tr> <tr> <td>ADD #data, ea</td> <td>ADDQ #data, ea</td> </tr> <tr> <td>SUB #data, ea</td> <td>SUBQ #data, ea</td> </tr> </tbody> </table> <p>where:</p> <p><i>data</i> Legal values are -128 to 127 for MOVE and MOVEQ instructions.  Legal values are 1 to 8 for ADD, ADDQ, SUB, and SUBQ instructions.</p> <p><i>ea</i> Effective address.</p> <p><i>n</i> Number of data register.</p> <p>For further information, see the <i>Motorola M68000 Microprocessor User's Manual</i></p>	Before the Change	After the Change	MOVE.L #data, Dn	MOVEQ #data, Dn	ADD #data, ea	ADDQ #data, ea	SUB #data, ea	SUBQ #data, ea	quick
Before the Change	After the Change									
MOVE.L #data, Dn	MOVEQ #data, Dn									
ADD #data, ea	ADDQ #data, ea									
SUB #data, ea	SUBQ #data, ea									

(cont.)

**Table 2-2. UNIX/DOS Assembler Command Line Flags (cont.)**

<b>Flags</b>	<b>Meaning (positive form)</b>	<b>Default</b>
rel32	<p>Forces the assembler to default to 32-bit base and outer displacements when the address range is 32 bits long. This flag can be turned on and off at any time in the program. It will only be effective if the processor type is set to a 32-bit processor (68020/30/40, 68EC020, 68EC030, 68EC040, and CPU32 family).</p> <p>Addressing modes which first appeared with the 68020:</p> <p>((bd,An,Xn) ([bd,An,Xn],od) ([bd,An],Xn,od)  (bd,PC,Xn) ([bd,PC,Xn],od) ([bd,PC],Xn,od))</p> <p>defaulted the size of the outer and base displacements to word for forward references, external references, complex expressions, and relocatable expressions. While the code produced was smaller, you had to always size cast displacement expressions if you had a processor that accessed a full 32-bit address range (68020/30/40 and CPU32 family). The rel32 flag lets you change the default to 32 bits without size casting displacement expressions.</p>	norel32
s	Lists the source text on the output listing.	s
t	Lists the symbol table on the output listing.	t
w	Prints warnings during the assembly.	w
x	Lists the cross-reference table on the output listing. The x option overrides the symbol table output option, t. If both t and x are used, a cross-reference table is generated.	nox

## Invocation Examples

The two examples that follow show how to enter various options on the command line. File name extensions in these examples are UNIX-specific and may be different for DOS.

### **Example:**

```
% asm68k -h -f fr1.p=68020 -l > multd.l multd.s
```

The source file `multd.s` is assembled. The `-l` option produces a listing file which is redirected from the standard output device to `multd.l`. The output object module generated by default is `multd.o`. The `-h` option causes production of an HP `asmb_sym` file named `multd.A`. The assembler will automatically set the `d` flag to place local symbols in the output object module. The `fr1` flag forces instructions that contain a forward reference to an absolute address to use a 32-bit address. The `p=68020` flag identifies the target processor as 68020.

### **Example:**

```
% asm68k -o temp.obj -f "now, case" divd.s
```

The input source `divd.s` is assembled. The output listing is not produced because the `-l` option is not used on the command line. The output object module is named `temp.obj`. The `flag` option `now` suppresses warning messages during assembly. The `flag` option `case` retains case sensitivity for symbols.

## LNK68K Linker

The Microtec Research LNK68K Linker combines relocatable object modules and libraries into a single absolute or relocatable module. In the process, it resolves memory references and resolves external references.

### Invocation Syntax

The following command line syntax shows how to invoke the LNK68K Linker. Note that the linker must be invoked with a command file or an input object file. The line continuation character, the backslash (\), continues the command line entry on the next line for UNIX systems only.

#### Syntax:

```
lnk68k [-c command_file] [-C command] [-F format]
        [-h] [-H link_symfile] [-l library]...
        [-m] [-M] [-o output_objectfile] [-r] [-u name] [-v]
        [input_object_file,input_object_file]...]
```

#### Description:

lnk68k	Specifies the name of the linker.
-c <i>command_file</i>	Identifies the named file as a command file, which contains one or more linker commands.
-C <i>command</i>	Specifies a linker command. If <i>command</i> requires spaces or any other special characters, it must be enclosed in quotes. The commands specified using -C are placed before any command specified in the command file or other command line option. Multiple linker commands can be specified by -C cmd1 -C cmd2, etc.
-F <i>format</i>	Specifies the format of the file created by the linker. Legal values for <i>format</i> are the same as those for the linker FORMAT command and include:
hp	HP-OMF format
ieee	IEEE-695 format (default)
incremental	IEEE-695 relocatable format (incremental linking)
S[n]	Indicates the Motorola S-record file format. <i>n</i> can be 1 or 2 (default).

	noabs	No object file is produced; however, internal processing is performed and a map file will be produced if requested.
-h		Produces an HP 64000-compatible absolute file (HP-OMF) and a <code>link_sym</code> file. Sets LISTABS PUBLICS and LISTABS INTERNALS.  The default file extension for the absolute file generated is .X. The <code>link_sym</code> file name is the source file root name with a .L extension.  For more information on this option, see the FORMAT HP linker command in the <i>Microtec Research ASM68K Reference Manual</i> . For more information on HP 64000 development support, see Appendix A.
-H <i>link_symfile</i>		Produces an HP 64000-compatible absolute file (HP-OMF) and a <code>link_sym</code> file. Sets LISTABS PUBLICS and LISTABS INTERNALS.  The default file extension for the absolute file generated is .X. The <code>link_sym</code> file name is the source file root name with a .L extension.  For more information on this option, see the FORMAT HP linker command in the <i>Microtec Research ASM68K Reference Manual</i> . For more information on HP 64000 development support, see Appendix A.
-l <i>library</i>		Specifies a library to be searched after all object files have been loaded. Multiple libraries can be specified as follows: -l file1.lib -l file2.lib.
-m		Writes a map file to the standard output device. The standard output can be redirected to a file.
-M		Indicates that a map file will be written to the file <code>command_file.map</code> or <code>input_object_file.map</code> if there is no command file ( <code>input_object_file</code> is the name of the first input object file specified on the command line).
-o [ <i>output_object_file</i> ]		Identifies the named file as the output object module. If this option is not specified, the default output file name is

*command\_file.extension* or, if a command file is not specified, the first *input\_object\_file.extension*, where *extension* is:

.o (UNIX)

.obj (DOS)

.x (UNIX)

.abs (DOS)

.X

If you use the -r option, the output object module is relocatable.

If -r, -H, or -h is not used, the output object module is absolute in either IEEE-695 or Motorola S-record format.

If you use the -H or -h option, the output object module is in HP format.

If -c *command\_file* is not specified, -o *output\_objectfile* must be specified to prevent overwriting the first input object file (due to the creation of the default output object module *input\_object\_file.extension*).

-r

Generates an incremental link. If this option is not specified, an absolute file is produced. This option is equivalent to the FORMAT INCREMENTAL linker command. Unless you have used the -c option, the -r option must be used with the -o option.

If you use the -r option, the output object module is relocatable. If -r is not used, the output object module is absolute in IEEE-695 or Motorola S-record format.

-u *name*

Creates an external reference for the linker to resolve. This option is equivalent to the EXTERN linker command.

-v

Displays the version number of LNK68K and then exits.

*input\_objectfile*

Specifies an input object module. The default file name extension is .o (UNIX) or .obj (DOS).

It is illegal to specify certain combinations of command line options. For example, it is illegal to specify both -r and -h. The linker writes an appropriate error message to the standard error device if an illegal combination is specified.

If you specify both *input\_object\_file* and the -c *command\_file* option:

- An absolute file (*command\_file.x* for UNIX, *command\_file.abs* for DOS) will be created in IEEE-695 format, unless a different format is specified

- No map file will be generated

If you only specify an *input\_object\_file*:

- An absolute file will be created in IEEE-695 format (*input\_object\_file.x* for UNIX, *input\_object\_file.abs* for DOS)
- No map file will be generated

If you only specify the **-c command\_file** option:

- An absolute file (*command\_file.x* for UNIX, *command\_file.abs* for DOS) will be created in IEEE-695 format, unless a different format is specified
- No map file will be generated

## Environment Variables

This section describes environment variables that affect LNK68K Linker operation. If you use these variables, they must be set before invoking the linker.

### MRI\_68K\_LIB

The **MRI\_68K\_LIB** environment variable specifies a directory path for files used by the **LOAD** command. The LNK68K Linker will use this path to search for libraries and object files. For more information, refer to the **LOAD** and **LOAD\_SYMBOLS** commands in the chapter *Linker Commands* in your *Reference Manual*.

#### Note

The **MRI\_68K\_LIB** environment variable can only be set for one directory path at a time. If you have another application that requires the **MRI\_68K\_LIB** environment variable to be set to a different directory path, you will have to reset this variable before using that application.

#### Examples:

```
setenv MRI_68K_LIB /my_lib_path      /* C shell (csh) */  
MRI_68K_LIB=/my_lib_path            /* Bourne shell (sh) */  
export MRI_68K_LIB  
  
SET MRI_68K_LIB=my_lib_path        /* DOS */
```

## TMP (DOS only)

The **TMP** environment variable specifies a directory path for temporary files:

```
SET TMP=my_tmpdir
```

### Example:

```
set TMP=D:\PATH
```

This command causes a subsequent invocation of LNK68K to use device D: and directory \PATH for temporary files.

The environment variable **TMP** should be set in your **AUTOEXEC.BAT** file. If this command is not specified or an invalid directory is used, the temporary files will be opened in the current directory.

Note that one useful definition for **TMP** would be to use your RAM disk, if you have such a pseudo-device.

## File Name Defaults

If file name extensions are not specified, the linker assumes the default file name extensions listed in Table 2-3. If a root name is not specified, the linker uses the root specified by the command line option **-c command\_file** or the first input object file name, in this order.

Table 2-3. UNIX/DOS Linker File Name Extensions

File	UNIX	DOS
Input command file	.cmd	.cmd
Input object file	.o	.obj
Relocatable object file	.o	.obj
IEEE-695 absolute object file (default)	.x	.abs
Motorola S-record absolute object file	.x	.abs
HP-OMF absolute object file	.X	.X
Link map file	.map	.map
Output HP-OMF link_sym file	.L	.L

The linker uses the current defaults if you do not specify a node, device, or directory.

## Continuation Character

A linker command can be continued to the next line by terminating it with one or more spaces followed by a pound sign (#). A continuation character must be placed like a comma between section names.

### Example:

```
ORDER SECT1, SECT2, SECT3, SECT4
```

This ORDER command can be written as:

```
ORDER SECT1 #
SECT2, SECT3 #
SECT4
```

## Invocation Examples

The following examples show various methods of invoking and using the linker. File name extensions in these examples are UNIX-specific and may be different for DOS.

### Invoking the Linker with a Command File

This method of invoking the linker lets you specify a command file or both a command file and additional object modules on the command line. Any object module files specified on the command line will be named in a LOAD command. This LOAD command will be inserted before the first LOAD command in the command file.

Command files let you set section addresses, define section loading order, and load a default set of object modules and libraries. Command files enter linker commands in batch mode. Any error in the command file is considered fatal by the linker and the load is not completed.

### Example:

```
* lmk68k -c lmult.cmd -m > lmult.map
```

The linker reads the command file lmult.cmd and the link map produced by the -m option is redirected (written) to the file lmult.map. The output object module produced is lmult.x and will be in IEEE-695 format unless a FORMAT command in the lmult.cmd file specified a different format.

**Example:**

```
% lnk68k -C "LISTMAP PUBLICS" -c sample.cmd -M -l lib1.lib  
mod1,mod2,mod3
```

The linker reads the command file **sample.cmd**, executes the commands in the file, and inserts three load commands which load the three object modules **mod1.o**, **mod2.o**, and **mod3.o**. The option **-C "LISTMAP PUBLICS"** will place this linker command at the beginning of the command file. The library **lib1.lib** is then searched to resolve any remaining unresolved external references in the loaded modules. The output object module is **sample**, and the output map file is **sample.map**.

If the file **sample.cmd** contains the following code:

```
ORDER SECT2,COMSEC  
PUBLIC ARG3=$3760  
* Load modules  
LOAD test1.o  
LOAD test2.o  
END
```

the command file processed by the linker will be:

```
LISTMAP PUBLICS  
ORDER SECT2,COMSEC  
PUBLIC ARG3=$3760  
* Load modules  
LOAD mod1.o  
LOAD mod2.o  
LOAD mod3.o  
LOAD test1.o  
LOAD test2.o  
LOAD lib1.lib  
END
```

**Invoking the Linker Without a Command File**

When there is no command file specified, the linker constructs a command file consisting of **LOAD** commands for the object module files.

**Example:**

```
% lnk68k -o sample -r -m mod1 mod2 mod3 > sample.map
```

The linker loads the three object modules **mod1.o**, **mod2.o**, and **mod3.o**. The **-r** option generates a relocatable object module for incremental linking: **sample.o**. The link map is redirected (written) to the file **sample.map**.

The command file processed by the linker will be:

```
LOAD mod1.o
LOAD mod2.o
LOAD mod3.o
END
```

**Example:**

```
% lnk68k -o abc.o -l def.lib -r -m >abc.map mod1.o mod2.o mod3.o
```

The linker loads the three object modules mod1.o, mod2.o, and mod3.o. The -r option generates a relocatable object module for incremental linking: abc.o. The link map is redirected (written) to the file abc.map.

The library def.lib is placed at the end of all the LOAD commands. The command file processed by the linker will be:

```
LOAD mod1.o
LOAD mod2.o
LOAD mod3.o
LOAD def.lib
END
```

## LIB68K Librarian

The Microtec Research LIB68K Object Module Librarian builds and maintains program libraries. Libraries are collections of relocatable object modules residing in a single file.

### Invocation Syntax

The command line can be continued on the next line if a minus sign (-) is the last character on the line to be continued.

Only certain library functions can be specified on the command line: **ADDMOD**, **DELETE**, **REPLACE**, **EXTRACT**, and **FULLDIR**. These librarian commands can be entered in any order, but the librarian processes the commands in the order specified above. Multiple object modules and files must be enclosed in double quotes.

#### Syntax:

```
lib68k [-a "filename[,filename]..."]  
        [-d "module_name[,module_name]..."]  
        [-e "module_name[,module_name]..."]  
        [-r "filename[,filename]..."]  
        [-l] [-v] library_file
```

#### Description:

**lib68k**                Invokes the librarian.

**-a *module\_name*[,*module\_name*]...**

                         Adds the named module(s) to the library. This option is equivalent to the **ADDMOD** library command.

**-d *module\_name*[,*module\_name*]...**

                         Deletes the named module(s) from the library. This option is equivalent to the **DELETE** library command.

**-e *module\_name*[,*module\_name*]...**

                         Extracts the named module(s) from the library. Specifically, the module(s) are copied to a file having the same name as *module\_name*. The modules are not deleted from the library. This option is equivalent to the **EXTRACT** library command.

**-l**

                         Lists the entire symbol table and module names on the standard output device. The standard output can be redirected to

a file. This option is equivalent to the FULLDIR library command.

**-r module\_name[,module\_name]...**

Replaces the named module(s) in the library with module(s) having the same name. This option is equivalent to the REPLACE library command.

**-v**

Displays the version number of LIB68K and then exits.

**library\_filename**      Specifies the library to be read or written.

The output will be displayed on your terminal unless you redirect your output to a file.

## File Name Defaults

A summary of librarian file name default extensions is shown in Table 2-4.

Table 2-4. UNIX/DOS Librarian File Name Extensions

File	UNIX	DOS
Input command file	.cmd	.cmd
Library file	.lib	.lib
Relocatable object module	.o	.obj
Listing file (as output from the internal librarian FULLDIR command)	.lst	.lst

The listing file contains output from the internal librarian FULLDIR command. This listing file should not be confused with the listing file named on the command line (using -l) which does not have a default file name extension.

Default file extensions are assumed when you do not append an extension. The librarian looks for a period (.) in the file name, scanning from the right to the left, and then compares the extension found to the default (.lib). If they are not the same, the librarian reports an error.

### Examples:

If the file name is lib.lib

open lib.	(this command fails)
open lib.l	(this command fails)
open lib.lib	(this command succeeds)
open lib	(this command succeeds)

## Invocation Examples

There are three ways to invoke the LIB68K Librarian: command line mode, command file mode, and interactive mode. The operating system prompt shown in the examples is denoted as a percent sign (%). Your system may be configured with a different prompt.

### Invoking the Librarian Interactively

You can enter librarian commands interactively from the terminal. A help facility is available by typing **h** or **help**.

#### Example:

```
lib68k
```

The librarian prompts you for commands (**LIB68K>**). If an illegal command is entered, the librarian displays an error message and provides an opportunity to reenter the command. In this interactive mode, most librarian command errors are not fatal. Multiple object modules or files must be enclosed within double quotes.

### Invoking the Librarian from the Command Line

You can enter librarian commands on the command line, but you can only modify an existing library using this method. To create a new library, you must run the librarian interactively or with a command file.

Only certain library functions can be specified on the command line: **ADDMOD**, **DELETE**, **REPLACE**, **EXTRACT**, and **FULLDIR**. These librarian commands can be entered in any order, but the librarian collects the commands and processes them in the order listed above.

#### Example:

```
% lib68k -r "sym1.o, sym2.o" -a "mod1.o, mod2.o, mod3.o" -l abc.lib
```

The **-r (REPLACE)** command replaces modules in abc.lib with modules of the same name contained in the files `sym1.o` and `sym2.o`. The **-a (ADDMOD)** command adds modules contained in the files `mod1.o`, `mod2.o`, and `mod3.o` to the library `abc.lib`. The contents of `abc.lib` are listed to standard output. Regardless of the order in which these commands are specified, the librarian will execute the additions followed by the replacements.

### Invoking the Librarian with a Command File

You can read librarian commands from a command file in batch mode. The commands are read in the exact order in which they are specified.

Any error encountered when processing the command file is printed, and execution of the command that generated the error is not completed. After the first error is encountered, commands are read, checked for errors, and executed if possible. However, if a library file is specified, it is not generated if an error is encountered.

**Example:**

```
% lib68k < command_file1
```

The librarian commands in *command\_file1* will be read in batch mode. Refer to the *Microtec Research ASM68K Reference Manual* for more details on the librarian commands.

## Utility Programs

### IEE2AOUT Conversion Program

The ASM68K Assembler generates object files in the Microtec Research extended IEEE-695 format. These files must be passed through a conversion utility before they can be input to a native linker or other *a.out*-consuming tool. This conversion utility converts the file into standard *a.out* or *b.out* format and is called *iee2aout*.

**Syntax:**

```
iee2aout [-b] [-mmagic] [-ms] [-N] ieefile outfile
```

**Description:**

<i>iee2aout</i>	Invokes the converter.
<i>-b</i>	Generates a file in <i>b.out</i> format instead of <i>a.out</i> . A <i>b.out</i> file contains more information in the file header and has a different magic number (0415).
<i>-m<sub>n</sub>magic</i>	Sets the magic number for the output file to <i>magic</i> . Valid values for <i>magic</i> include any decimal, octal, or hexadecimal constant. (default magic number: 0407)
<i>-ms</i>	Sets the machine type in the magic number for the Sun host. This option is only valid if you are running on a Sun workstation.
<i>-N</i>	Uses OMAGIC (0407) instead of ZMAGIC (0413) for the executable output file. When ZMAGIC is used, the file header is mapped onto the text section.

- ieefefile*      Specifies the assembler file to be converted.  
*outfile*      Specifies the name of the converted object module.

### Section Typing

The IEEE section names for **text**, **data**, and **bss** sections are shown in Table 2-5.

Table 2-5. IEEE Section Names

<b>text</b>		<b>data</b>		<b>bss</b>
.text	text	.data	iports	.bss
const	strings	reserved	tags	bss
initfini	code	?INITDATA	vars	zerovars
9	literals	14	heap	
		data	stack	

Sections that have names not shown in Table 2-5 are considered text or data according to the IEEE section type. For example, **ROM** and **CODE** sections are treated as **text** and **DATA** sections are treated as **data**. Sections with no IEEE type are treated as **text** with **b.out** and **data** with **a.out**.

### Relocation File Conversion

When generating an IEEE object module to be converted by the **iee2aout**, utility, the object module can only have names for three sections: **text**, **data**, and **bss**. Section information can be rearranged as long as the three section limit is maintained. The section names can be renamed using the following compiler options:

- NC (rename **const** variable section name)
- NI (rename initialized data section name)
- NL (rename compiler-generated literals section name)
- NS (rename string section name)
- NT (rename code section name)
- NZ (rename **zerovars** section name)

#### Example:

```
-NCdata -NIdata -NLdata -NSdata -NTtext -NZbss
```

This example shows section renaming to maintain the three section limit. These compiler options are correct if no external references are required to be resolved by the IEEE libraries.

#### Example:

```
-NCtext -NIdata -NLtext -NSText -NTtext -NZbss
```

This example includes constants, strings, and literals in the text section.

### Absolute File Conversion

Absolute IEEE files are the normal output of the IEEE linker. iee2aout can handle multiple sections of different types, but all sections of the same a.out type (i.e., text, data, and bss) must be grouped together. Your linker command should contain the following lines:

```
order      code,literals,strings,const    ;text sections
order      vars,tags,ioports,heap,stack   ;data sections
order      zerovars                      ;bss sections
sectsize heap=heapsize                 ;heap size
```

Both the heap and the stack are in the data section. Modify your start-up code to reassign heap and stack locations.

### Creating Your own Library

When you want to use library routines not provided by the Microtec Research library, you must:

1. Establish which Microtec Research library functions are needed to resolve implicit function calls (these are calls that the compiler makes to underlying functions; the function calls are not present in your source code). To determine the functions needed, identify the unresolved variables by:
  - a. Linking with a non-Microtec Research library to see what is unresolved
  - b. Using the FULLDIR librarian command to see where those variables reside

2. Create your own library that contains the Microtec Research library modules that you want to use by invoking the librarian and performing the following commands:

```
create mylib.lib
addlib Microtec_Research_library (mod1,mod2,...)
save
quit
```

where *Microtec\_Research\_library* indicates the name of the Microtec Research library and *mod1, mod2, ...* represent the Microtec Research library modules that you want. For more information on library commands, refer to Chapter 13, *Librarian Commands*, in your *ASM68K Reference Manual*.

3. Compile and assemble your files as usual.
4. Perform an incremental link to create an object module with the three a.out sections (see the previous section *Section Typing*), whose only unresolved symbols are defined in libraries not provided by Microtec Research.

```
lnk68k -r -o ieee_out -c merge.cmd ieeel_in ieee2_in
mylib.lib
```

The merge.cmd linker command file contains the following three lines:

```
merge text code
merge data vars, strings, const, literals
merge bss zerovars
```

5. Convert the resulting object file to a.out or b.out format.

#### Example:

```
mcc68k -c -g -o file1.ie1 file1.c
mcc68k -c -g -o file2.ie1 file2.c
lnk68k -r -o file.ie2 -c merge.cmd file1.ie1 file2.ie1
mylib.lib
iee2aout file.ie2 file.o
```

The source files file1.c and file2.c are compiled, producing object files that are linked according to the merge.cmd linker command file to produce the relocatable output file file.ie2. This file is then converted to a.out format and named file.o.

**Note**

Avoid incrementally linking one module at a time to prevent modules in your library from being multiply defined.

## Return Codes

The assembler, linker, and librarian pass a return code to the operating system upon completion of program execution. The return code indicates whether the program executed properly or not. The return codes are:

- 0      Warning or No Error.

Either the program ran to completion with no user errors (No Error) or there were user-created warnings (Warning).

- 1      Execution Error or Fatal Error.

Either the program ran to completion with user-created assembler, linker, or librarian errors (Execution Error) or the program did not run to completion due to a system problem (Fatal Error). For example, the program was not allocated the required amount of disk space, or a file read/write error occurred.

A message describing the problem will accompany the return code for a warning, execution error, or fatal error.

## Introduction

This chapter describes the basic use of the assembler, linker, and object module librarian for use on a VAX host computer running the VMS operating system.

The chapter is divided into sections, one for each tool; each section contains descriptions of the following information:

- Invocation syntax
- Environment variables, if any
- Input and output file name defaults
- Command line flag descriptions, if any
- Invocation examples

The last sections of the chapter describe the operating system return codes generated by the tools and utility programs that come with the assembler package. The operating system prompt used in the directions is a dollar sign (\$). Your system may be configured with a different prompt.

## ASM68K Assembler

The Microtec Research ASM68K Assembler converts assembly language programs to relocatable object code. Object modules are produced in IEEE-695 format. After assembly, object modules are combined by the linker.

## LNK68K Linker

The Microtec Research LNK68K Linker allows multiple relocatable object modules to be combined into a single relocatable module that subsequently can be re-linked with other modules (i.e., incremental linking). The linker can also combine relocatable object modules into a single absolute object module in one of the following formats:

- Motorola S-record format
- IEEE-695 format
- Hewlett-Packard 64000 format (HP-OMF)

The LNK68K Linker has some powerful features designed for embedded applications, which provide the capability to load your software at locations that match your unique hardware environment. You can specify:

- Where to load segments
- Order to load segments
- Size of segments to use

If one or more of the linker input files is a library of object modules, the linker automatically loads all modules from the library that are referenced by the individually named object modules.

## **LIB68K Librarian**

The Microtec Research LIB68K Object Module Librarian lets you maintain a collection of relocatable object modules that reside in one file. By using the librarian, you can format and organize library files that will subsequently be used by the linker.

## **ASM68K Assembler**

The Microtec Research ASM68K Assembler produces object code typically for later use by the LNK68K Linker and LIB68K Librarian.

### **Invocation Syntax**

This section describes the command line syntax needed to invoke the ASM68K Assembler. Abbreviations for options are shown in the *Description* section. After entering an abbreviation, any number of characters can be specified up to and including the option's full spelling.

Use parentheses to specify multiple settings for the same option. For example, `/IPATH=(/dir1,/dir2)` will search both directories, as specified. However, if you had entered `/IPATH=dir1 /IPATH=dir2`, only the last parameter is used; the first directory is not searched. Another example, `/DEFINE=(A=10, B=20)`, will set the symbols A and B to 10 and 20, respectively.

Options are case-insensitive and are separated by a slash (/). To preserve upper- and lower-case distinctions, place the specified text between double quotes ("").

**Syntax:**

```
A68K [ /DEFINE=sym[=value] ] [ /FLAGS="flag_list" ]
[ /HP | /NOHP ] [IPATH=dir] [ /LIST[=list_file] | /NOLIST ]
[ /OBJECT[=object_file] | /NOBJECT ]
[ /VERSION] source_file
```

**Description:****DEFINE=sym[=value]**

Defines the symbol *sym* at assembly time. The parameter *value* must be a constant. If *value* is not specified, *sym* is set to 1.

The value of *sym* will be overridden by the first SET directive applied to *sym* in the source file. No EQU directives can be applied to *sym*.

**FLAGS="flag\_list"** Enables and disables the assembler command line flags (see Table 3-2). This option may be specified before or after the *source\_file*.

**HP** Produces an HP 64000 compatible (HP-OMF) *asmb\_sym* symbol file. The D flag is automatically set, placing local symbols in the output object module. The default file name is *source\_file.A*. This option may be specified before or after the *source\_file*.

(abbreviation = H; default = NOHP)

For more information, see Appendix A, *HP 64000 Development System Support*.

**NOHP** Does not produce an HP 64000 compatible (HP-OMF) *asmb\_sym* file. This option may be specified before or after the *source\_file*.

(default = NOHP)

**IPATH=dir** Specifies a path to be searched to locate files that are to be included using the INCLUDE directive. For more information on the INCLUDE directive, see *Assembler Directives* in the *Microtec Research ASM68K Reference Manual*.

The assembler first searches for include files in the directory containing the assembler source program. It next searches the directory specified by *dir*. A maximum of four paths can be specified.

LIST[=list_file]	Produces a listing output file with the file name indicated. This option may be specified before or after the <i>source_file</i> . (abbreviation = /LIS, default = /LIST= <i>source_file</i> .LIS)
NOLIST	Inhibits production of a listing file. This switch overrides directives in the assembler source file. This option may be specified before or after the <i>source_file</i> .
OBJECT[=object_file]	Produces an output object module with the file name indicated. This option may be specified before or after the <i>source_file</i> . (abbreviation = /OBJ, default = /OBJECT= <i>source_file</i> .OBJ)
NOBJECT	Inhibits production of an output object module. This switch overrides directives in the assembler source file. This option may be specified before or after the <i>source_file</i> .
VERSION	Displays the version number of ASM68K and then exits.
<i>source_file</i>	Specifies the assembler input file.

If you do not specify options, the following defaults will be used:

```
/NOHP
/LIST=source_file.LIS
/OBJECT=source_file.OBJ
```

Refer to Table 3-2 for the default flags that will be used.

## File Name Defaults

If *source\_file* does not contain an extension, the assembler assumes the extensions listed in Table 3-1. Specification of the object file and list file are optional. An output object module is produced by default, but if no output file name is specified on the command line, the assembler uses *source\_file* as the root file name and appends the default file name extensions listed in Table 3-1.

Table 3-1. VMS Assembler File Name Extensions

File	Extension
Assembler source file	.SRC
HP-OMF asmb_sym file	.A
Listing file	.LIS
Relocatable object file	.OBJ

File names can include some or all of the following specifications:

- Node name (current default assumed)
- Device name (current default assumed)
- Directory (current default assumed)
- File name
- Extension
- Version number

## Command Line Flags

Flags entered on the command line control the assembly. Assembler command line flags for *flag\_list* in the **FLAGS** command line option are listed in Table 3-2.

Abbreviations for the flags are also shown. After entering an abbreviation, any number of characters can be specified up to the flag's full spelling. Some flags may be negated by using the prefix **no** or a minus sign (-). Multiple items in the list are separated by commas, and enclosed within double quotes. Items requiring the use of parentheses must be enclosed in double quotes. Spaces cannot appear in the list unless the list is enclosed within double quotes. If conflicting flags are specified or a flag is specified multiple times, the last flag specified overrides all previous conflicting flags.

More information about the flags can be obtained in the description of the **OPT** directive in the *Microtec Research ASM68K Reference Manual*. Flags entered on the command line have the effect of inserting an **OPT** directive at the beginning of the source file. However, flags on the command line do not override other **OPT** directives in the source file.

Table 3-2. VMS Assembler Command Line Flags

Flag	Meaning	Default
abspcadd	When this flag is set, an absolute expression in conjunction with the mnemonic <b>PC</b> (i.e., <b>5(PC)</b> ) refers to an absolute address accessed through PC relative mode rather than to a relative displacement to the current program counter.  For more information, refer to <i>Assembler Syntax for Effective Address Fields</i> in Chapter 3 of your <i>ASM68K Reference Manual</i> .	abspcadd
brb		
brs		
b	Forces forward references in relative branch instructions ( <b>Bcc</b> , <b>BRA</b> , <b>BSR</b> ) to use the short form of the instruction (8-bit displacement). If <b>nobrl</b> , <b>nobrs</b> , or <b>nob</b> is indicated, the assembler will be set to the default <b>brw</b> .	brw
brl	Forces forward references in relative branch instructions ( <b>Bcc</b> , <b>BRA</b> , <b>BSR</b> ) to use the long address (32 bits). If <b>OPT OLD</b> is in effect in conjunction with the <b>brl</b> flag, the longer form is used. If <b>nobrl</b> is indicated, the assembler be set to the default <b>brw</b> . This command is effective only on 32-bit chips (68EC020/030/040, 68020/30/40, and CPU32 family).  For 32-bit processors, when <b>OPT OLD</b> has not been specified, 32-bit displacements are used. When <b>OPT OLD</b> has been specified, 16-bit displacements are used. In all other processor modes, 16-bit displacements are used.	brw
brw	Forces 16-bit displacements to always be used in relative branch instructions ( <b>Bcc</b> , <b>BRA</b> , <b>BSR</b> ) that have forward references. <b>brw</b> cannot be negated.	brw

(cont.)

Table 3-2. VMS Assembler Command Line Flags (cont.)

Flag	Meaning	Default
case ca	Retains case-sensitivity of symbols. For example, <b>abc</b> and <b>ABC</b> are treated as different symbols.	case
cex c	Lists all lines of object code that are generated by the <b>DC</b> directive. The <b>nocex</b> flag only lists the first line of a <b>DC</b> directive.	cex
cl i	Lists instructions that are not assembled due to conditional assembly statements.	cl
cre	Lists the cross-reference table on the output listing. This option overrides the symbol table output option, <b>t</b> . If both <b>t</b> and <b>cre</b> are used, a cross-reference table is generated.	nocre
d	Places local symbols in the output object module. This flag is automatically set if you generate an <b>asmb_sym</b> file by using the <b>HP</b> option.	nod
e	Lists lines with errors on your terminal as well as on the output listing.	e
frl	Forces instructions that contain a forward reference to an absolute (non-relocatable) address to use a 32-bit address instead of a 16-bit address. This command is effective only on 32-bit chips (68EC020/030/040, 68020/30/40, and CPU32 family).	32-bit address or <b>frl</b>
frs f	Forces instructions that contain a forward reference to an absolute (nonrelocatable) address to use a 16-bit address instead of a 32-bit address.	nofrs
g	Lists assembler-generated symbols in the symbol or cross-reference table. If <b>d</b> is also specified, these symbols are included in the object module as well.	nog

(cont.)

Table 3-2. VMS Assembler Command Line Flags (cont.)

Flag	Meaning	Default
llen= <i>n</i>	Sets the length of the line on the source listing to be <i>n</i> columns long. The <i>n</i> can be a number between 37 and 1120, inclusive.	llen=132
mc	Prints macro calls on the program listing.	mc
md	Prints macro definitions on the program listing.	md
mex	Lists macro and structured control directive expansions on the program listing.	mex
m		
nest= <i>n</i>	Sets <i>n</i> nesting levels for macros. The maximum number of nesting levels allowed is 100.	nest=100
o	Produces the output object module.	o
old	Specifies that the interpretation of the <b>b1</b> flag and the <b>L</b> size qualifiers be 16-bit displacements for Bcc instructions when <b>OPT BRL</b> or <b>OPT -B</b> are specified, or when explicit <b>L</b> qualifiers are used in 68020 mode. This is useful when migrating 68000 programs to 32-bit processors (68EC020/030/040, 68020/30/40, and CPU32 family).	old
op= <i>n</i>	Resets the maximum number of optimization loops that the assembler will perform if the <b>opnop</b> flag is set. The assembler will discontinue looping if there is a pass in which no optimization occurs.	op=3

(cont.)

Table 3-2. VMS Assembler Command Line Flags (cont.)

Flag	Meaning	Default
opnop	Removes for optimization purposes any NOPs that have been generated by the assembler because of forward references. The assembler will often assume the worst case in determining the size of forward-referenced variables and will reserve the maximum space (usually 32 bits) for them. During the second pass of the assembler, NOPs will be generated if the operand represented by the variable can be used in an instruction format that is more compact than the size reserved during the first pass.	noopnop
p= <i>type[/cotype]</i> . . .	<p>Specifies a set of compatible processors and coprocessors. The <i>type</i> can be any processor of the 68000 family. A list of target processors and instruction sets is shown in Table 5-2 in the <i>Reference Guide</i>.</p> <p>The <i>cotype</i> can be either 68881 or 68851 with the following restrictions for <i>type</i>:</p> <ul style="list-style-type: none"> <li>• If the <i>cotype</i> is 68881, the 68040 processor <i>type</i> is illegal.</li> <li>• If the <i>cotype</i> is 68851, the 68030 and 68040 processor <i>types</i> are illegal.</li> </ul> <p><b>Example:</b></p> <p><i>p=68020/68881/68851</i></p> <p>It is an error to specify an incompatible set of processors, such as <i>p=68030/68851</i>.</p>	68000/68881
pco p	Uses the program counter relative addressing mode on backward references within an absolute (ORG) section whenever possible. This flag is only applied to instructions where the relative addressing mode is legal and the displacement from the program counter fits within the 16-bit field provided.	nopco

(cont.)

Table 3-2. VMS Assembler Command Line Flags (cont.)

Flag	Meaning	Default								
pcr	<p>Specifies that a PC-relative address mode will be used on references from a relocatable section to the same section. This applies to all instructions for which the PC-relative address mode is legal.</p> <p>The pcr flag differs from pcs in that pcs applies to all relocatable sections within this module and assumes you know the boundaries. The pcr flag only affects intrasection expressions in the current module (expressions for which the assembler has enough information to generate the correct offset).</p>	pcr								
pcs r	<p>Uses the PC-relative address mode on backward references from a relocatable section to the same section or to a different relocatable section. This address mode will be used for all instructions for which the PC-relative address mode is legal.</p>	nopcs								
quick	<p>Changes the MOVE, ADD, and SUB instructions to the more efficient MOVEQ, ADDQ, and SUBQ instructions when possible.</p> <table> <thead> <tr> <th>Before the Change</th> <th>After the Change</th> </tr> </thead> <tbody> <tr> <td>MOVE.L #data, Dn</td> <td>MOVEQ #data, Dn</td> </tr> <tr> <td>ADD #data, ea</td> <td>ADDQ #data, ea</td> </tr> <tr> <td>SUB #data, ea</td> <td>SUBQ #data, ea</td> </tr> </tbody> </table> <p>where:</p> <ul style="list-style-type: none"> <li><i>data</i> Legal values are -128 to 127 for MOVE and MOVEQ instructions. Legal values are 1 to 8 for ADD, ADDQ, SUB, and SUBQ instructions.</li> <li><i>ea</i> Effective address.</li> <li><i>n</i> Number of data register.</li> </ul> <p>For further information, see the <i>Motorola M68000 Microprocessor User's Manual</i></p>	Before the Change	After the Change	MOVE.L #data, Dn	MOVEQ #data, Dn	ADD #data, ea	ADDQ #data, ea	SUB #data, ea	SUBQ #data, ea	quick
Before the Change	After the Change									
MOVE.L #data, Dn	MOVEQ #data, Dn									
ADD #data, ea	ADDQ #data, ea									
SUB #data, ea	SUBQ #data, ea									

(cont.)

Table 3-2. VMS Assembler Command Line Flags (cont.)

Flag	Meaning	Default
rel32	Forces the assembler to default to 32-bit base and outer displacements when the address range is 32 bits long. This flag can be turned on and off at any time in the program. It will only be effective if the processor type is set to 68020 or 68030. It will only be effective if the processor type is set to a 32-bit processor (68020/30/40, 68EC020, 68EC030, 68EC040, and CPU32 family).  Addressing modes which first appeared with the 68020:  ((bd,An,Xn) ([bd,An,Xn],od) ([bd,An],Xn,od) (bd,PC,Xn) ([bd,PC,Xn],od) ([bd,PC],Xn,od))  defaulted the size of the outer and base displacements to word for forward references, external references, complex expressions, and relocatable expressions. While the code produced was smaller, you had to always size cast displacement expressions if you had a processor that accessed a full 32-bit address range (68020/30/40 and CPU32 family). The rel32 flag lets you change the default to 32 bits without size casting displacement expressions.	norel32
s	Lists the source text on the output listing.	s
t	Lists the symbol table on the output listing.	t
w	Prints warnings during the assembly.	w
x	Lists the cross-reference table on the output listing. This option overrides the symbol table output option, t. If both t and x are used, a cross-reference table is generated.	nox

## Invocation Examples

The examples in this section show how to invoke the assembler and specify multiple flags on the command line.

### Example:

```
$ A68K MOD68K
```

The assembler assembles the source file **MOD68K.SRC** and produces the default output listing **MOD68K.LIS** and output object module **MOD68K.OBJ**.

### Example:

```
$ A68K/FLAGS="CRE,G" MULTD
```

The assembler assembles the source file **MULTD.SRC** and produces the default output listing **MULTD.LIS** and the output object module **MULTD.OBJ**. The **CRE** and **G** options force a cross-reference table to be printed on the output listing and assembler-generated symbols to be included in the cross-reference table.

### Example:

```
$ A68K/FLAGS="-C,P=68020"/NOLIST/OBJECT=TEMP DIVD
```

The assembler assembles the source file **DIVD.SRC** and produces the output object module **TEMP.OBJ**. Generation of an output listing is inhibited by the **NOLIST** option. The **-C** option prevents the listing after the first line of all lines of object code that are generated by the **DC** directive. The **P=68020** assembles code for the 68020 or equivalent microprocessor.

### Example:

```
$ A68K/HP MOD1/FLAGS="P=68020,X"
```

The assembler assembles the source file **MOD1.SRC** and produces the output listing **MOD1.LIS**, and the relocatable object module **MOD1.OBJ**. The **/HP** option causes the **asmb\_sym** file **MOD1.A** to be produced as well. The assembler automatically sets the **D** flag, which includes local symbols in the output object module. The **X** flag includes the output listing on the cross-reference table. The **P=68020** flag defines assembled code for the 68020 microprocessor.

## LNK68K Linker

The Microtec Research LNK68K Linker combines relocatable object modules and libraries into a single absolute or relocatable module. In the process, it relocates memory references and resolves external references.

## Invocation Syntax

The following command line syntax shows how to invoke the LNK68K Linker. Note that the linker must be invoked with a command file or an input object file.

Abbreviations for options are also shown. After entering the abbreviation, any number of characters can be specified up to and including the option's full spelling.

Use parentheses to specify multiple settings for the same option. For example, /COMMAND=(cmd1,cmd2) will execute two linker commands, as specified. However, if you had entered /COMMAND=cmd1 /COMMAND=cmd2, only the last parameter is used; the first command is not executed.

Options are case-insensitive and separated by a slash (/).

### Syntax:

```
L68K [ /ABSOLUTE[=absolute_file] | /NOABSOLUTE ]
      [ /COMMAND=command ] [ /FORMAT=format ]
      [ /HP | /NOHP ] [ /LIBRARY=library ]
      [ /MAP=map_file | /NOMAP ] [ /OBJECT=relocatable_file ]
      [ /REFERENCE=name] [ /VERSION ]
      { /OPTION command_file | object_file[,object_file]... }
```

### Description:

**L68K** Invokes the linker.

**ABSOLUTE[=*absolute\_file*]**

Produces an absolute object file with the file name indicated.  
(abbreviation = ABS, default = *command\_file.ABS*)

**NOABS** Inhibits generation of an absolute object module.

**COMMAND=*command***

Specifies a linker command. This command behaves as though it is placed at the top of the *command\_file*. Multiple linker commands can be specified.

**FORMAT=*format***

Specifies the format of the file created by the linker. Legal values for *format* are the same as those for the linker **FORMAT** command and include:

<b>HP</b>	HP-OMF format
-----------	---------------

<b>IEEE</b>	IEEE-695 format (default)
-------------	---------------------------

## INCREMENTAL

	IEEE-695 relocatable format (incremental linking)
S	Motorola S-record
NOABS	No object file is produced; however, internal processing is performed and a map file will be produced if requested.
HP	Produces HP 64000-compatible object modules (HP-OMF) and <code>link_sym</code> files. Sets <code>LISTABS PUBLICS</code> and <code>LISTABS INTERNALS</code> .
	The default file extension for the absolute file generated is <code>.X</code> . The <code>link_sym</code> file is given the source file root name with a <code>.L</code> extension.
	For more information on this option, see the <b>FORMAT HP</b> linker command in the <i>Microtec Research ASM68K Reference Manual</i> . For more information on HP 64000 development support, see Appendix A.
NOHP	Does not produce an HP 64000 compatible (HP-OMF) <code>asmb_sym</code> file. This option may be specified before or after the <i>source file</i> .
LIBRARY= <i>library</i>	Specifies a library to be searched after all object files have been loaded.
MAP [ <i>=map_file</i> ]	Produces a link map with the file name indicated. The default file name is the name of the source file with the extension <code>.MAP</code> . (abbreviation = <code>M</code> , default = <i>source_file.MAP</i> )
NOMAP	Does not produce an output link map.
OBJECT[ <i>=relocatable_file</i> ]	Produces a relocatable object file that can be linked with other object modules. <i>relocatable_file</i> must be specified in batch mode. (abbreviation = <code>OBJ</code> )
	This option can be used in place of <code>/ABSOLUTE</code> to incrementally link object module files. Since an absolute link is

not performed, unresolved external references to other modules can exist in the resulting relocatable object module.

If you use this option with **/FLAGS=B** on the command line or the **LIST B** option in the command file, you will generate absolute output instead of relocatable output.

For more information on this option, see the **FORMAT INCREMENT** linker command in the *Microtec Research ASM68K Reference Manual*.

**OPTION** *command\_file*

Indicates use of a batch command file.  
(abbreviation = **OP**)

**REFERENCE=name** Creates an external reference called *name* for the linker to resolve. For more information on this option, see the **EXTERN** linker command in the *Microtec Research ASM68K Reference Manual*.

**VERSION** Displays version number of LNK68K and then exits.

**object\_file** Specifies the source object file(s).

If you do not specify options, the following defaults will be used:

```
/OBJECT=source_file.OBJ  
/LIST=source_file.LIS  
/NOHP
```

Refer to the *Invocation Examples* section for examples of the interactive and non-interactive linker invocation modes.

It is illegal to specify certain combinations of command line options. For example, it is illegal to specify both **ABS** and **OBJ**, or **OBJ** and **HP**. The linker displays an appropriate error message if an illegal combination is specified.

## File Name Defaults

Specification of the map and output object module file names is optional. If they are not specified, the linker produces them by default. If an output file name is not specified, the linker uses either the *command\_file*, if present, or the first *object\_file*, as the root file name for generated files. If file name extensions are not specified, the default extensions listed in Table 3-3 are assumed.

**Table 3-3. VMS Linker File Name Extensions**

File	Extension
Input object file	.OBJ
Input command file	.OPT
Relocatable output object file	.OBJ
Absolute object file	.ABS
HP-OMF absolute object file	.X
Link map file	.MAP
HP 64000 link_sym file	.L

The linker uses the current defaults if you do not specify a node, device, or directory.

## Invocation Examples

The following examples show various methods of invoking the linker. Examples include multiple flags on the command line, interactive command input, incremental linking, and flags overriding other specified flags.

### Invoking the Linker with a Command File

You can enter linker commands in batch mode through the use of a command file. Any fatal error in the command file prevents the link from being completed. However, all commands in the file are processed and checked for errors.

#### Example:

```
$ L68K/OPTION CMD/MAP=MY.MAP
```

The linker reads the command file **CMD.OPT** and produces a link map called **MY.MAP**. The absolute object file **CMD.ABS** is generated in IEEE-695 format.

#### Example:

```
$ L68K/OPTION/HP LMOD1.CMD
```

The command file **LMOD1.CMD** is read, and the link map produced is named **LMOD1.MAP**.

The **LISTABS PUBLICS** and **LISTABS INTERNALS** linker commands are automatically set by the use of the **/HP** option, so the linker includes external and local symbols in the output file. The **/HP** option generates the absolute object module, **LMOD1.X** in **HP-OMF** format, and the **HP link\_sym** file, **LMOD1.L**. External and local symbols are placed in the output object module.

**Example:**

```
$ L68K/HP/OPT LFILE
```

The command file **LFILE.OPT** is read, and the link map produced is named **LFILE.MAP**. If an extension had been specified with the command file name, then that command file with the specified extension would have been read. The **/HP** option causes generation of the absolute object module **LFILE.X** in **HP-OMF** format and the **HP link\_sym** file **LFILE.L**.

The **LISTABS PUBLICS** and **LISTABS INTERNALS** linker commands, which include both local and external symbols in the output object module, are automatically set by the linker.

### Invoking the Linker from the Command Line

You can enter the names of object modules that will be linked together on the command line. You cannot enter linker commands that perform functions such as adjusting segment addresses or defining public symbols. However, any such commands encountered in a command file will be processed. When this invocation method is used, the object modules are linked starting at the default address of zero, the default segment order is used, and the standard link map is generated.

**Example:**

```
$ L68K/OBJECT=LORP/NOMAP DORP, FORP, SORP
```

The object modules **DORP.OBJ**, **FORP.OBJ**, and **SORP.OBJ** are incrementally linked forming the relocatable object module output file **LORP.OBJ**. Output of a link map is inhibited by use of the **/NOMAP** option.

## Continuation Character

A linker command can be continued to the next line by terminating it with one or more spaces followed by a pound sign (#). A continuation character must be placed like a comma between section names.

### Examples:

```
ORDER SECT1,SECT2,SECT3,SECT4
```

This ORDER command can be written as:

```
ORDER SECT1 #
      SECT2,SECT3 #
      SECT4
```

## LIB68K Librarian

The Microtec Research LIB68K Librarian formats and organizes files into program libraries so the linker can automatically load frequently used object modules without concern for the specific names and characteristics of the modules.

### Invocation Syntax

The command line can be continued on the next line if a minus sign (-) is the last character on the line to be continued.

Only certain library functions can be specified on the command line: **ADDMOD**, **DELETE**, **REPLACE**, **EXTRACT**, and **FULLDIR**. These librarian commands can be entered in any order, but the librarian processes the commands in the order specified above. Multiple object modules and files must be enclosed in double quotes.

### Syntax:

```
B68K  [/OPTION command_file | library_file ]
      [/ADDMOD="module_name" [, "module_name"]...]
      [/DELETE="module_name" [, "module_name"]...]
      [/EXTRACT="module_name" [, "module_name"]...]
      [/FULLDIR[=library_listfile]] [/OUTPUT=list_file]
      [/REPLACE="file_name" [, "file_name"]...]
      [/VERSION]
```

**Description:**

B68K                    Invokes the librarian.

OPTION *command\_file*

Indicates use of a command file, which contains linker commands. In the interactive mode, the command file is the terminal (TT:).  
(abbreviation = OP)

*library\_file*        Specifies the library file to be read or written.

ADDMOD= "*module\_name*" [ , "*module\_name*" ] ...

Adds the named module(s) to the library. This option is equivalent to the **ADDMOD** library command.

DELETE= "*module\_name*"

Deletes the named module(s) from the library. This option is equivalent to the **DELETE** library command.

EXTRACT= "*module\_name*"

Extracts the named module(s) from the library. Specifically, the module(s) are copied to a file having the same name as *module\_name* with the extension .OBJ. The modules are not deleted from the library. This option is equivalent to the **EXTRACT** library command.

FULLDIR[=*library\_listfile*]

Displays the contents of the library named on the command line (*library\_file*). If the file *library\_listfile* is not specified, the library information is written to the terminal.

OUTPUT= *list\_file*      Specifies the output listing file.

REPLACE= "*file\_name*"

Replaces the named module(s) in the library with module(s) having the same name. This option is equivalent to the **REPLACE** library command.

VERSION

Displays the version number of LIB68K and then exits.

## File Name Defaults

Table 3-4 lists the librarian file name extension defaults.

**Table 3-4. VMS Librarian File Name Extensions**

File	Extension
Library file	.LIB
Listing file	.LIS
Relocatable object module	.OBJ
Input command file	.OPT

## Invocation Examples

There are three ways to invoke the LIB68K Librarian: interactive mode, command line mode, and command file mode.

### Invoking the Librarian Interactively

You can enter librarian commands interactively from the terminal. A help facility is available by typing **h** or **help**. When an illegal command is entered, the librarian displays an error message and provides an opportunity to re-enter the command. In this interactive mode, most librarian command errors are not fatal. Multiple object modules or files must be enclosed within double quotes.

#### Example:

```
$ B68K
```

The librarian displays a prompt (**LIB68K>**) for interactive input.

### Invoking the Librarian from the Command Line

You can enter librarian commands on the command line. This method can only be used to modify an existing library. A new library cannot be created using this method.

Only certain library functions can be specified using this method; these are **ADDMOD**, **DELETE**, **REPLACE**, **EXTRACT**, and **FULLDIR**. These librarian commands can be entered in any order, but the librarian collects the commands and processes them in a fixed order: **ADDMOD**, **DELETE**, **REPLACE**, **EXTRACT**, and **FULLDIR**. **FULLDIR** implies **DIRECTORY**.

**Example:**

```
$ B68K EXLIB.LIB/REPLACE="SYM1.OBJ,SYM2.OBJ" -  
$ /ADDMOD="MOD1.OBJ,MOD2.OBJ,MOD3.OBJ"
```

The REPLACE option replaces modules SYM1.OBJ and SYM2.OBJ in the library file EXLIB.LIB. The ADDMOD option adds modules MOD1.OBJ, MOD2.OBJ, and MOD3.OBJ to the library EXLIB.LIB. The librarian will execute the ADDMOD command, then the REPLACE command. These commands can be specified on the command line in any order. The command line continuation character (-) continues the command line entry on a second line.

**Invoking the Librarian with a Command File**

You can read librarian commands from a command file in batch mode. The commands are read in the exact order in which they are specified. If an error is found, commands read after the first error is found are processed, checked for errors, and executed if possible. If a library file is specified, it is not generated if an error is encountered.

**Example:**

```
$ B68K /OPTION COMMAND.OPT /OUTPUT=PRINTOUT
```

The library commands are all contained in the command file COMMAND.OPT. The output listing will be written to the file PRINTOUT.LIS.

## Utility Programs

This section provides a description of the command line syntax and instructions for using the utility and conversion programs supplied with ASM68K. These programs are:

IEE2AOUT              Converts IEEE files to a.out format.

### IEE2AOUT Conversion Utility

The ASM68K Assembler generates object files in the Microtec Research extended IEEE-695 format. These files must be passed through a conversion utility before

they can be input to a native linker or a.out-consuming tool. This conversion utility converts the file into standard a.out or b.out format and is called iee2aout.

### Note

On VMS systems, iee2aout is a foreign image. Once you enter the DCL command:

```
iee2aout ::= $ sys$common:[sysexe]iee2aout.exe
```

iee2aout can be invoked using the syntax shown.

### Syntax:

```
iee2aout [-b] [-mmagic] [-ms] [-N] ieefile outfile
```

### Description:

<i>iee2aout</i>	Invokes the converter.
-b	Generates a file in b.out format instead of a.out. A b.out file contains more information in the file header and has a different magic number (0415).
-m <i>magic</i>	Sets the magic number for the output file to <i>magic</i> . Valid values for <i>magic</i> include any decimal, octal, or hexadecimal constant. (default magic number: 0407)
-ms	Sets the machine type in the magic number for the Sun host. This option is only valid if you are running on a Sun workstation.
-N	Uses OMAGIC (0407) instead of ZMAGIC (0413) for the executable output file. When ZMAGIC is used, the file header is mapped onto the text section.
<i>ieefile</i>	Specifies the assembler file to be converted.
<i>outfile</i>	Specifies the name of the converted object module.

## Section Typing

The IEEE section names for **text**, **data**, and **bss** sections are shown in Table 3-5.

**Table 3-5. IEEE Section Names**

<b>text</b>		<b>data</b>		<b>bss</b>
.text	text	.data	iports	.bss
const	strings	reserved	tags	bss
initfini	code	?INITDATA	vars	
9	literals	14	heap	zerovars
		data	stack	

Sections that have names not shown in Table 3-5 are considered text or data according to the IEEE section type. For example, **ROM** and **CODE** sections are treated as **text** and **DATA** sections are treated as **data**. Sections with no IEEE type are treated as **text** with **b.out** and **data** with **a.out**.

## Relocation File Conversion

When generating an IEEE object module to be converted by the **iee2aout**, utility, the object module can only have names for three sections: **text**, **data**, and **bss**. Section information can be rearranged as long as the three section limit is maintained. The section names can be renamed using the following compiler options:

**/rename=(const=name)** to rename **const** variable section

**/rename=(initvars=name)** to rename initialized data section

**/rename=(literals=name)** to rename compiler-generated literals section

**/rename=(strings=name)** to rename string section

**/rename=(code=name)** to rename code section

**/rename=(zerovars=name)** to rename **zerovars** section

### Example:

```
/rename=(const=data) /rename=(initvars=data)
/rename=(literals=data) /rename=(strings=data)
/rename=(code=text) /rename=(zerovars=bss)
```

This example shows section renaming to maintain the three section limit. These compiler options are correct if no external references are required to be resolved by the IEEE libraries.

**Example:**

```
/rename=(const=text) /rename=(initvars=data)
/rename=(literals=text) /rename=(strings=text)
/rename=(code=text) /rename=(zerovars=bss)
```

This example includes constants, strings, and literals in the text section.

**Note**

If your target does not support operations encountered in the source file, such as floating-point conversion or long division, the compiler generates function calls to perform these operations.

**Absolute File Conversion**

Absolute IEEE files are the normal output of the IEEE linker. iee2aout can handle multiple sections of different types, but all sections of the same a.out type (i.e., *text*, *data*, and *bss*) must be grouped together. Your linker command should contain the following lines:

```
order      code,literals,strings,const    ;text sections
order      vars,tags,ioports,heap,stack   ;data sections
order      zerovars                      ;bss sections
sectsize heap=heapsize                 ;heap size
```

Both the heap and the stack are in the data section. Modify your start-up code to reassigned heap and stack locations.

**Creating Your own Library**

When you want to use library routines not provided by the Microtec Research library, you must:

1. Establish which Microtec Research library functions are needed to resolve implicit function calls (these are calls that the compiler makes to underlying functions; the function calls are not present in your source code). To determine the functions needed, identify the unresolved variables by:
  - a. Linking with a non-Microtec Research library to see what is unresolved
  - b. Using the FULLDIR librarian command to see where those variables reside

2. Create your own library that contains the Microtec Research library modules that you want to use by invoking the librarian and performing the following commands:

```
create mylib.lib
addlib Microtec_Research_library (mod1, mod2, ...)
save
quit
```

where *Microtec\_Research\_library* indicates the name of the Microtec Research library and *mod1, mod2, ...* represent the Microtec Research library modules that you want. For more information on library commands, refer to Chapter 13, *Librarian Commands*, in your *ASM68K Reference Manual*.

3. Compile and assemble your files as usual.
4. Perform an incremental link to create an object module with the three **a.out** sections (see the previous section *Section Typing*), whose only unresolved symbols are defined in libraries not provided by Microtec Research.

```
168k /format=incremental /object=ieee_out
      /option merge.cmd
```

A sample *merge.cmd* linker command file could contain the following lines:

```
merge text code
merge data vars, strings, const, literals
merge bss zerovars
load inputfile1
load inputfile2
load mylib.lib
```

5. Convert the resulting object file to **a.out** or **b.out** format.

#### Example:

```
c68k /debug file1.c
c68k /debug file2.c
a68k /object=file1.ie1 file1.src
a68k /object=file2.ie1 file2.src
168k /format=incremental /object=file.ie2
      /option mymerge.cmd
iee2aout file.ie2 file.o
```

The source files `file1.c` and `file2.c` are compiled, producing assembly files that are assembled and linked according to the `mymerge.cmd` linker command file:

```
merge text code
merge data vars, strings, const, literals
merge bss zerovars
load file1.ie1
load file2.ie1
load mylib.lib
```

to produce the relocatable output file `file.ie2`. This file is then converted to `a.out` format and named `file.o`.

#### Note

Avoid incrementally linking one module at a time to prevent modules in your library from being multiply defined.

## Creation of Temporary Files

The linker and assembler can create temporary files by calling the `tmpfile()` function from the VAX C library. `tmpfile()` generates a temporary file based on the process identification number and should open a temporary file in the `SYSSSCRATCH` VAX directory. The user can redirect the temporary file to a disk or another destination by using the following syntax:

```
SYSSSCRATCH = "DISK$USER:[destination]"
```

#### Example:

```
SYSSSCRATCH = "DISK$USER:[MYDISK]"
```

In this example, the temporary file generated in `SYSSSCRATCH` was redirected to `MYDISK`.

## VMS Return Codes

The assembler, linker, and librarian pass a return code to VMS upon completion of program execution. The return code indicates whether the program executed properly or not. The return codes are:

- 0      Warning.  
The program generated valid output, but it issued warnings to indicate that the output may not be what you expected.
- 1      No Error.  
The program executed properly with no user errors.
- 2      Execution Error.  
The program executed properly, but there were user-created assembler, linker, or librarian errors.
- 4      Fatal Error.  
The program did not execute properly due to a system problem. For example, the program was not allocated the required amount of disk space, or a file read/write error occurred.



# HP 64000 Development System Support: Appendix A

---

## Overview

LNK68K can generate of HP 64000 object modules and debugging files that can be downloaded and used with the HP 64000 Development System or HP 64700 family emulator. The following development system instruments can be used:

- HP 64700 Family Emulator
- HP 64243 Emulator
- HP 64620 Logic State/Software Analyzer
- HP 64310 Software Performance Analyzer

You can generate the object module in HP-OMF format and the associated symbol files by using the **-h**/ **-H** options (UNIX/DOS) or the **/HP** option (VMS) on the assembler and linker command lines. For the assembler, these options cause generation of the **asmb\_sym** file. When you use these options, the assembler automatically sets the **debug** flag, forcing local symbols to be included in the output object file. The linker automatically executes the **LISTABS PUBLICS** and **LISTABS INTERNALS** linker commands to include both external and local symbols in the output file and generates a **link\_sym** file.

For more information on the HP 64000 Development System, see the *Section Types and HP 64000 Symbolic Files* section in Chapter 4, *Relocation*, of the *ASM68K Reference Manual*.

## HP 64000 Considerations

When using ASM68K for program development in conjunction with the HP 64000 development system, you should be aware of several inherent limitations of the HP 64000. These limitations are:

- The HP 64000 limits symbol names to 15 characters. Therefore, the assembler and linker truncate symbols to 15 characters and issue diagnostics for symbols whose first 15 characters are not unique.
- Symbols cannot include the characters ?, ., or \$. These characters are translated to underscores ( \_ ) by the assembler or are translated by the

linker in the case of library routines. This translation can result in duplicate symbol errors.

- Sections on the HP 64000 are classified as **PROG**, **DATA**, **COMN**, or **ABSOLUTE**. ASM68K sections must be mapped into these sections. ASM68K provides an option on the **SECT** assembler directive that allows the specification of the section type. These types are shown in Table A-1 along with how they are mapped into HP 64000 sections.

**Table A-1. Section Types for the HP 64000 Sections**

ASM68K Section Type	HP 64000 Section
C (CODE)	PROG
D (DATA)	DATA
R (ROM)	COMN

If the section type is not explicitly defined, the assembler will attempt to make its own decision about how the section is used. It will attempt to place the code section in **PROG**, the uninitialized data section in **DATA**, and the initialized data section in **COMN**.

If multiple segments containing instructions exist, and no segments containing data definition directives exist, the **DATA** and **COMN** sections will be assigned these "extra" segments containing instructions. If multiple segments containing data definition directives exist, and no segments containing instructions exist, the **PROG** section will be assigned one of these "extra" data segments. Any remaining segments will be mapped into absolute sections.

If there are multiple code sections or multiple data sections defined in ASM68K, the second and successive sections are mapped into absolute sections. Absolute sections do not have local symbol and line number information associated with them; therefore, when debugging, you cannot reference source lines in absolute sections.

The following considerations apply when using the HP 64000 development system (but do not apply for the 64700 family emulator):

- The HP 64000 uses two special symbols in each routine (**Rlabel** and **Elabel**) as special identifiers for measurement tools. These symbols represent the return point and end address of a routine. The *label* portion of the name is the same as the routine name. These labels are generated automatically by the Microtec Research MCC68K Compiler when the h

switch is specified at compile time. For programs written in assembly language, you must manually encode these labels.

- The HP 64000 microprocessor monitor program must be linked in with your code. Therefore, in order to use the 68000 family monitor program, it must be uploaded to your host computer, assembled with ASM68K, and linked with the application program by LNK68K. However, before the monitor can be assembled, it has to be modified to account for incompatibilities between the HP assembly language and the Microtec Research/Motorola assembly language. The modifications required are minor and are summarized in Table A-2.

**Table A-2. Modifications Required for the 68000 Monitor**

Hewlett-Packard Directive/Instruction	Microtec Research Directive/Instruction
*68000*	TTL *68000*
GLB SYMBOL	XDEF SYMBOL
SKIP	PAGE
CMP.L #MONITOR_START,2(A7)	CMP.L #MONITOR_START,2(A7)
MOVE [SP]+,PSTATUS	MOVE (SP)+,PSTATUS
ASC 'Error! Entry Mode=User'	DC.B 'Error! Entry Mode=User'
DC.W REG_END-\$	DC.W REG_END-*
PROG	SECTIONPROG



# Index

---

## Symbols

- command line continuation character 2-21,  
    3-21  
# continuation character 2-18, 3-18

## A

-a librarian command line option 2-21  
a.out 2-24, 3-22  
ABSOLUTE linker command line option 3-13  
abspcadd assembler command line flag 2-6,  
    3-6  
ADDMOD librarian command 2-21  
ADDMOD librarian command line option 3-19

### Assembler

    description 1-1  
    UNIX/DOS 2-2 to 2-12  
        file name defaults 2-4  
        flags 2-5 to 2-11  
            abspcadd 2-6  
            brb 2-6  
            brl 2-6  
            brs 2-6  
            brw 2-6  
            case 2-7  
            cex 2-7  
            cl 2-7  
            cre 2-7  
            d 2-7  
            e 2-7  
            frl 2-7  
            g 2-7  
            i 2-7  
            llen 2-8  
            mc 2-8  
            md 2-8  
            mex 2-8  
            nest 2-8  
            o 2-8

old 2-8  
op 2-8  
opnop 2-9  
p 2-9  
pco 2-9  
pcr 2-10  
pcs 2-10  
quick 2-10  
r 2-10  
rel32 2-11  
s 2-11  
t 2-11  
w 2-11  
x 2-11  
invocation examples 2-12  
invocation syntax 2-2  
options 2-2 to 2-4  
    -b 2-2  
    -D 2-2  
    -f 2-3  
    -h 2-3  
    -I 2-3  
    -L 2-3  
    -l 2-3  
    -o 2-3  
    -V 2-3

VAX/VMS 3-2 to 3-12  
    file name defaults 3-4  
    flags 3-5 to 3-11  
        abspcadd 3-6  
        brb 3-6  
        brl 3-6  
        brs 3-6  
        brw 3-6  
        case 3-7  
        cex 3-7  
        cl 3-7  
        cre 3-7  
        d 3-7  
        e 3-7

frl 3-7  
 frs 3-7  
 g 3-7  
 i 3-7  
 llen 3-8  
 mc 3-8  
 md 3-8  
 mex 3-8  
 nest 3-8  
 o 3-8  
 old 3-8  
 op 3-8  
 opnop 3-9  
 p 3-9  
 pco 3-9  
 pcr 3-10  
 pcs 3-10  
 quick 3-10  
 r 3-10  
 rel32 3-11  
 s 3-11  
 t 3-11  
 w 3-11  
 x 3-11  
**invocation examples** 3-12  
**invocation syntax** 3-2 to 3-3  
**options** 3-3 to 3-4  
 DEFINE 3-3  
 FLAGS 3-3  
 HP 3-3  
 IPATH 3-3  
 LIST 3-4  
 NOHOP 3-3  
 NOLIST 3-4  
 NOOBJECT 3-4  
 OBJECT 3-4  
 VERSION 3-4

**B**

-b assembler command line option 2-2  
 b.out 2-24, 3-22  
 brb assembler command line flag 2-6, 3-6  
 brl assembler command line flag 2-6, 3-6

brs assembler command line flag 2-6, 3-6  
 brw assembler command line flag 2-6, 3-6

**C**

-C linker command line option 2-13  
 -c linker command line option 2-13  
 case assembler command line flag 2-7, 3-7  
 cex assembler command line flag 2-7, 3-7  
 cl assembler command line flag 2-7, 3-7  
**Command file**  
     -c linker option 2-13  
**Command line continuation character**  
     linker 3-21  
**Command line flags**  
     assembler  
         UNIX/DOS 2-5 to 2-11  
         VAX/VMS 3-5 to 3-11  
**COMMAND** linker command line option 3-13  
**Components of the ASM68K package** 1-1  
**Continuation character (#)** 2-18, 3-18  
**Continuation on command line**  
     minus sign 2-21, 3-18  
**Conventions, notational** viii  
**Conversion programs**  
     IEE2AOUT 2-24 to 2-28, 3-21 to 3-26  
**cre** assembler command line flag 2-7, 3-7

**D**

d assembler command line flag 2-7, 3-7  
 -D assembler option 2-2  
 -d librarian command line option 2-21  
 Data flow program diagram 1-3  
 DEFINE assembler command line option 3-3  
 DELETE librarian command 2-21, 3-19  
 DELETE librarian command line option 3-19  
 DOS  
     (see UNIX/DOS)

**E**

e assembler command line flag 2-7, 3-7  
 -e librarian command line option 2-21  
**Environment variables**

- MRI\_68K\_LIB 2-16  
TMP 2-5, 2-17  
EXTERN linker command 2-15  
EXTRACT librarian command 2-21, 3-19  
EXTRACT librarian command line option 3-19
- F**
- f assembler command line option 2-3  
-F linker command line option 2-13  
File formats  
    a.out 2-24, 3-22  
    b.out 2-24, 3-22  
    HP-OMF 2-13, 3-13  
    IEEE-695 2-13, 3-13  
    S-record 2-13, 3-13  
File name defaults  
    assembler  
        UNIX/DOS 2-4  
        VAX/VMS 3-4  
    librarian  
        UNIX/DOS 2-22  
        VAX/VMS 3-20  
    linker  
        UNIX/DOS 2-17  
        VAX/VMS 3-16
- FLAGS assembler command line option 3-3  
FORMAT linker command line option 3-13  
frl assembler command line flag 2-7, 3-7  
frs assembler command line flag 2-7, 3-7  
FULLDIR librarian command 2-21, 3-19  
FULLDIR librarian command option 3-19
- G**
- g assembler command line flag 2-7, 3-7
- H**
- h assembler command line option 2-3  
-H linker command line option 2-14  
-h linker command line option 2-14  
HP 64000  
    considerations A-2
- limitations A-1  
linker support A-1  
support A-1  
HP assembler command line option 3-3  
HP linker command line option 3-14
- I**
- i assembler command line flag 2-7, 3-7  
-I assembler command line option 2-3  
IEE2AOUT conversion utility 2-24 to 2-28, 3-21 to 3-26
- Invocation examples**
- UNIX/DOS**
- assembler 2-12  
    librarian 2-23 to 2-24  
        command file 2-23 to 2-24  
        command line 2-23  
        interactive 2-23  
    linker 2-18 to 2-20  
        command file 2-18  
        without command file 2-19 to 2-20
- VAX/VMS**
- assembler 3-12  
        command line 3-12  
        multiple flags 3-12  
    librarian 3-20 to 3-21  
        command file 3-21  
        command line 3-20  
    linker 3-16 to 3-17  
        command file 3-16  
        command line 3-17
- VMS**
- linker  
        command file 3-16
- Invocation syntax**
- assembler**
- UNIX/DOS 2-2  
    VAX/VMS 3-2 to 3-3
- librarian**
- UNIX/DOS 2-21  
    VAX/VMS 3-18
- linker**
- UNIX/DOS 2-13

- VAX/VMS 3-13
- IPATH assembler command line option 3-3
- L**
  - L assembler command line option 2-3
  - l assembler command line option 2-3
  - l librarian command line option 2-21
  - l linker command line option 2-14
  - Librarian
    - description 1-2
    - UNIX/DOS 2-21 to 2-24
      - file name defaults 2-22
      - invocation examples 2-23 to 2-24
      - options 2-21 to 2-22
        - a 2-21
        - d 2-21
        - e 2-21
        - l 2-21
        - r 2-22
        - V 2-22
    - VAX/VMS 3-18 to 3-21
      - file name defaults 3-20
      - invocation examples 3-20 to 3-21
      - invocation syntax 3-18
      - options 3-19
        - ADDMOD 3-19
        - DELETE 3-19
        - EXTRACT 3-19
        - FULLDIR 3-19
        - OPTION 3-19
        - OUTPUT 3-19
        - REPLACE 3-19
        - VERSION 3-19
  - LIBRARY linker command line option 3-14
  - Linker
    - command line continuation character 3-21
    - description 1-1
    - UNIX/DOS 2-13 to 2-20
      - file name defaults 2-17
      - invocation examples 2-18 to 2-20
      - options 2-13 to 2-16
        - C 2-13
        - c 2-13
  - F 2-13
  - H 2-14
  - h 2-14
  - I 2-14
  - M 2-14
  - m 2-14
  - o 2-14
  - r 2-15
  - u 2-15
  - V 2-15
- VAX/VMS 3-12 to 3-18
  - file name defaults 3-16
  - invocation examples 3-16 to 3-17
  - invocation syntax 3-13
  - options 3-13 to 3-15
    - ABSOLUTE 3-13
    - COMMAND 3-13
    - FORMAT 3-13
    - HP 3-14
    - LIBRARY 3-14
    - MAP 3-14
    - NOABS 3-13
    - NOHP 3-14
    - NOMAP 3-14
    - OBJECT 3-14
    - OPTION 3-15
    - REFERENCE 3-15
    - VERSION 3-15
  - LIST assembler command line option 3-4
  - llen assembler command line flag 2-8, 3-8
- M**
  - M linker command line option 2-14
  - m linker command line option 2-14
  - MAP linker command line option 3-14
  - mcc assembler command line flag 2-8, 3-8
  - md assembler command line flag 2-8, 3-8
  - mex assembler command line flag 2-8, 3-8
  - MRI\_68K\_LIB environment variable 2-16
  - Multiple flags, entering
    - assembler 2-5, 3-12

## N

nest assembler command line flag 2-8, 3-8  
NOABS linker command line option 3-13  
NOHP assembler command line option 3-3  
NOHP linker command line option 3-14  
NOLIST assembler command line option 3-4  
NOMAP linker command line option 3-14  
NOOBJECT assembler command line option  
    3-4  
Notational conventions viii

## O

-o assembler command line flag 2-8, 3-8  
-o assembler command line option 2-3  
-o linker command line option 2-14  
OBJECT assembler command line option 3-4  
Object file  
    -o assembler option 2-3  
OBJECT linker command line option 3-14  
old assembler command line flag 2-8, 3-8  
op assembler command line flag 2-8, 3-8  
opnop assembler command line flag 2-9, 3-9  
OPTION librarian command line option 3-19  
OPTION linker command line option 3-15  
Output file  
    -o assembler option 2-3  
OUTPUT librarian command line option 3-19

## P

p assembler command line flag 2-9, 3-9  
pco assembler command line flag 2-9, 3-9  
pcr assembler command line flag 2-10, 3-10  
pcs assembler command line flag 2-10, 3-10

## Q

Questions viii  
quick assembler command line flag 2-10, 3-10

## R

r assembler command line flag 2-10, 3-10

-r librarian command line option 2-22  
-r linker command line option 2-15  
REFERENCE linker command line option  
    3-15  
rel32 assembler command line flag 2-11, 3-11  
REPLACE librarian command 2-22  
REPLACE librarian command line option 3-19  
Return codes  
    UNIX/DOS 2-28  
    VAX/VMS 3-27

## S

s assembler command line flag 2-11, 3-11  
Suggestions viii  
SYS\$SCRATCH 3-26

## T

t assembler command line flag 2-11, 3-11  
Temporary file creation  
    VAX/VMS 3-26  
Temporary file placement  
    DOS 2-5  
TMP environment variable 2-5, 2-17  
tmpfile() VAX function 3-26

## U

-u linker command line option 2-15  
UNIX  
    (see UNIX/DOS)  
UNIX/DOS 2-1 to 2-28  
    assembler 2-2 to 2-12  
        command line flags 2-5 to 2-11  
            abspcadd 2-6  
            brb 2-6  
            brl 2-6  
            brs 2-6  
            brw 2-6  
            case 2-7  
            cex 2-7  
            cl 2-7  
            cre 2-7  
            d 2-7

e 2-7  
frl 2-7  
frs 2-7  
g 2-7  
i 2-7  
llen 2-8  
mc 2-8  
md 2-8  
mex 2-8  
nest 2-8  
o 2-8  
old 2-8  
op 2-8  
opnop 2-9  
p 2-9  
pco 2-9  
pcr 2-10  
pcs 2-10  
quick 2-10  
r 2-10  
rel32 2-11  
s 2-11  
t 2-11  
w 2-11  
x 2-11  
command line options 2-2 to 2-4  
    -b 2-2  
    -D 2-2  
    -f 2-3  
    -h 2-3  
    -I 2-3  
    -L 2-3  
    -l 2-3  
    -o 2-3  
    -V 2-3  
file name defaults 2-4  
invocation examples 2-12  
invocation syntax 2-2  
introduction 2-1  
librarian 2-21 to 2-24  
    command line options 2-21 to 2-22  
        -a 2-21  
        -d 2-21  
-e 2-21  
-I 2-21  
-r 2-22  
-V 2-22  
file name defaults 2-22  
invocation examples 2-23 to 2-24  
linker 2-13 to 2-20  
    command line options 2-13 to 2-16  
        -C 2-13  
        -c 2-13  
        -F 2-13  
        -H 2-14  
        -h 2-14  
        -o 2-14  
        -r 2-15  
        -u 2-15  
        -V 2-15  
    file name defaults 2-17  
    illegal option combinations 2-15  
    invocation examples 2-18 to 2-20  
    return codes 2-28  
    utility programs  
        IEE2AOUT 2-24 to 2-28  
Utility programs  
    IEE2AOUT 2-24 to 2-28, 3-21 to 3-26

V

-V assembler command line option 2-3  
-V linker command line option 2-15  
-V librarian command line option 2-22  
VAX  
    (see VAX/VMS)  
VAX/VMS 3-1 to 3-27  
    assembler 3-2 to 3-12  
        command line flags 3-5 to 3-11  
            abspcadd 3-6  
            brb 3-6  
            brl 3-6  
            brs 3-6  
            brw 3-6  
            case 3-7  
            cex 3-7  
            cl 3-7

cre .3-7  
d 3-7  
e 3-7  
fri 3-7  
frs 3-7  
g 3-7  
i 3-7  
llen 3-8  
mc 3-8  
md 3-8  
mex 3-8  
nest 3-8  
o 3-8  
old 3-8  
op 3-8  
opnop 3-9  
p 3-9  
pco 3-9  
pcr 3-10  
pcs 3-10  
quick 3-10  
r 3-10  
rel32 3-11  
s 3-11  
t 3-11  
w 3-11  
x 3-11  
command line options 3-3 to 3-4  
  DEFINE 3-3  
  FLAGS 3-3  
  HP 3-3  
  IPATH 3-3  
  LIST 3-4  
  NOHP 3-3  
  NOLIST 3-4  
  NOOBJECT 3-4  
  OBJECT 3-4  
  VERSION 3-4  
file name defaults 3-4  
invocation examples 3-12  
invocation syntax 3-2 to 3-3  
introduction 3-1  
invocation syntax 3-18  
librarian 3-18 to 3-21  
  command line options 3-19  
    ADDMOD 3-19  
    DELETE 3-19  
    EXTRACT 3-19  
    FULLDIR 3-19  
    OPTION 3-19  
    OUTPUT 3-19  
    REPLACE 3-19  
    VERSION 3-19  
  file name defaults 3-20  
  invocation examples 3-20 to 3-21  
linker 3-12 to 3-18  
  command line options 3-13 to 3-15  
    ABSOLUTE 3-13  
    COMMAND 3-13  
    FORMAT 3-13  
    HP 3-14  
    LIBRARY 3-14  
    MAP 3-14  
    NOABS 3-13  
    NOHP 3-14  
    NOMAP 3-14  
    OBJECT 3-14  
    OPTION 3-15  
    REFERENCE 3-15  
    VERSION 3-15  
  file name defaults 3-16  
  illegal option combinations 3-15  
  invocation examples 3-16 to 3-17  
  invocation syntax 3-13  
  return codes 3-27  
  temporary files 3-26  
  utility programs  
    IEE2AOUT 3-21 to 3-26  
VERSION assembler command line option 3-4  
VERSION librarian command line option 3-19  
VERSION linker command line option 3-15  
VMS  
  (see VAX/VMS)  
**W**  
w assembler command line flag 2-11, 3-11

**X**

x assembler command line flag 2-11, 3-11

XRAY Debugger

description 1-2

## Reader's Response

Please complete and return the following Reader's Response form to help us improve our documentation. Your comments and suggestions are always appreciated.

Product Name \_\_\_\_\_

Your Name \_\_\_\_\_

Company \_\_\_\_\_

Phone Number \_\_\_\_\_

Manual Date/Version \_\_\_\_\_

Title \_\_\_\_\_

Address \_\_\_\_\_

What is your level of programming experience?

Advanced

Intermediate

Beginner

In this language?

Advanced

Intermediate

Beginner

Circle the number which best expresses your rating of the information in this manual (*5 is the highest rating*):

Complete?	1	2	3	4	5	Easy to understand?	1	2	3	4	5
-----------	---	---	---	---	---	---------------------	---	---	---	---	---

Accurate?	1	2	3	4	5	Helpful examples?	1	2	3	4	5
-----------	---	---	---	---	---	-------------------	---	---	---	---	---

Well organized?	1	2	3	4	5	Helpful procedures?	1	2	3	4	5
-----------------	---	---	---	---	---	---------------------	---	---	---	---	---

Easy to Find?	1	2	3	4	5	At an appropriate level?	1	2	3	4	5
---------------	---	---	---	---	---	--------------------------	---	---	---	---	---

Easy to Read?	1	2	3	4	5	Overall rating?	1	2	3	4	5
---------------	---	---	---	---	---	-----------------	---	---	---	---	---

3.) Which programming language do you use?

---

4.) Does the manual provide all the necessary information? If not, what is missing?

---

5.) Are more examples needed? If so, where?

---

6.) Did you have any difficulty understanding descriptions or wording? If so, where?

---

7.) Which sections of this manual are the most important? The most helpful?

---

8.) What are the best features of this manual?

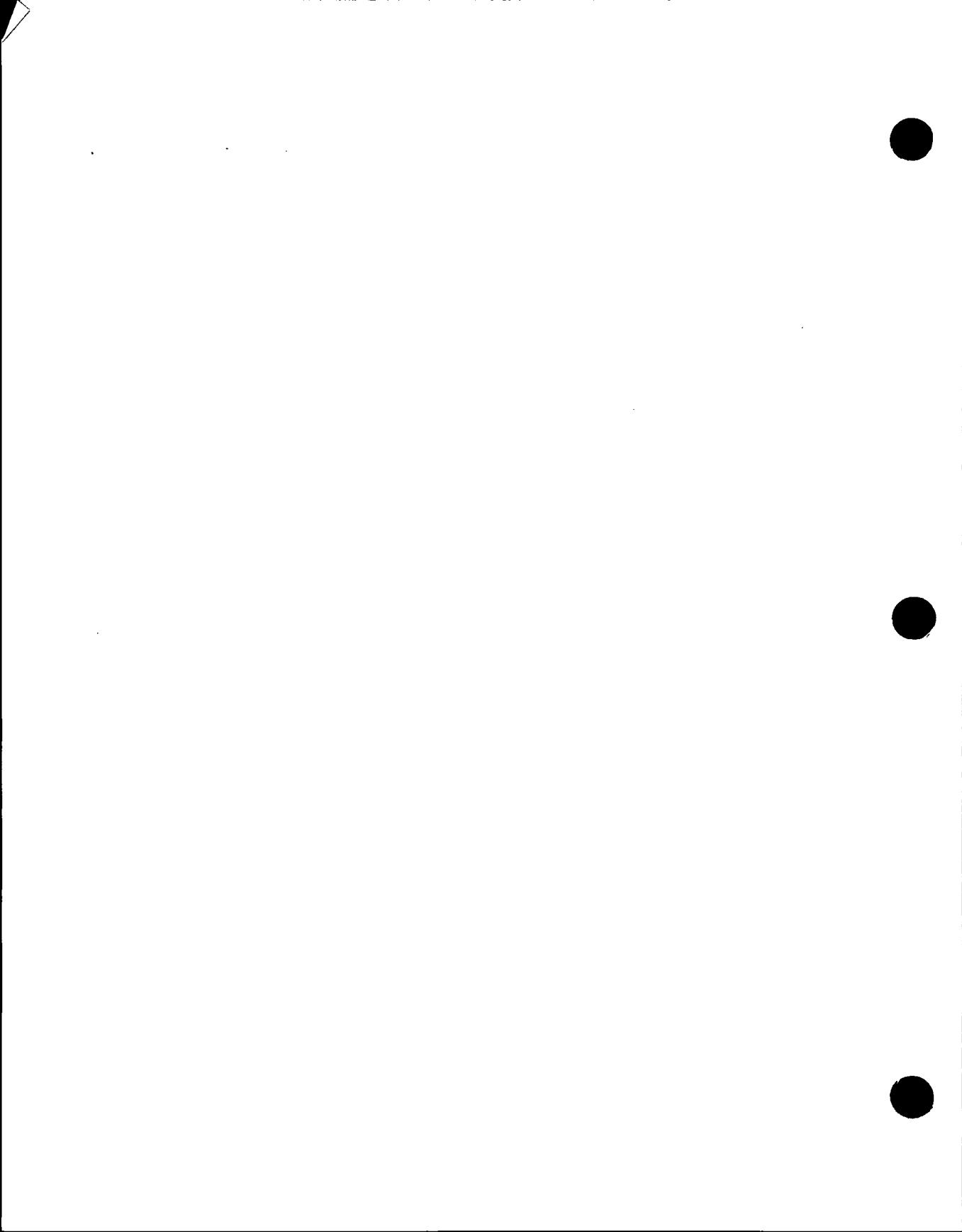
---

9.) What are the areas in this manual that need improvement?

---

10.) Did you find any errors in this manual? (Specify section and page number.)

---



## **Software Performance Report Instructions**

If you encounter problems or errors when using Microtec Research software products (including documentation), you can send a Software Performance Report (SPR):

- Electronically through e-mail to support@mri.com

or

- By completing the SPR form on the next page and returning it to:

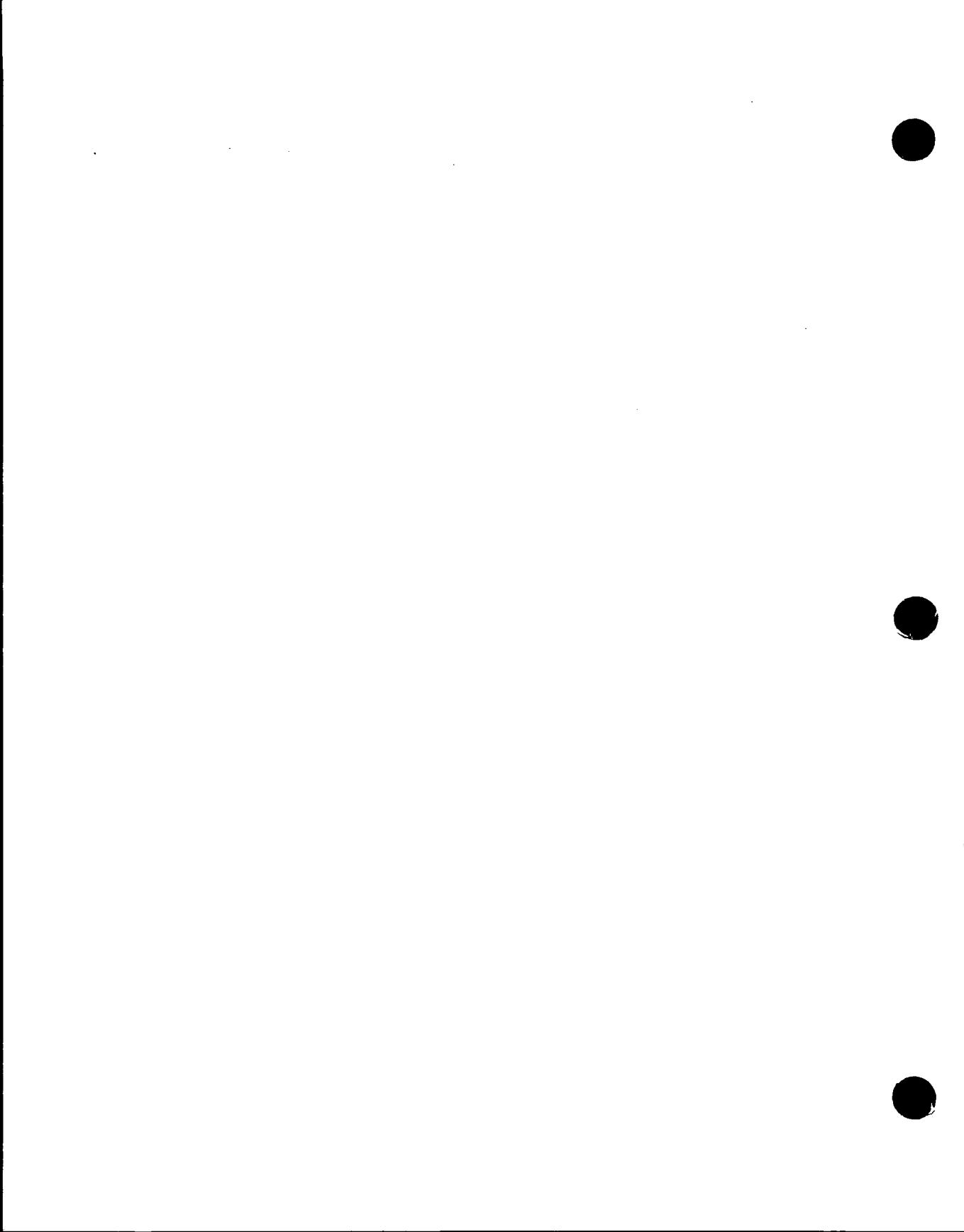
**Microtec Research  
Software Performance Reports  
2350 Mission College Boulevard  
Santa Clara, CA 95054**

All SPRs are directed to our Technical Support Department for analysis and response.

To make it easier for us to respond quickly to your problem, please include the following:

1. Give a complete description of the problem, error, or suggestion, including the exact wording of any error messages.
2. If possible, isolate the problem by providing a small example.
3. If you are mailing or FAXing an SPR and your error example is longer than one page of source code, please provide all information on a tape or floppy disk.
4. Include command files, listings, load maps, include files, data files, and all other relevant material with the message.
5. If applicable, send sample input and output listings.
6. If the output is from a compiler, send a mixed (source/assembly code) listing.
7. Be sure to include the following information:
  - Product name and version number
  - Serial number
  - Name of contact
  - Company purchased from
  - Company name and a shipping address
  - Phone number and/or FAX number
  - Host machine
  - Host operating system and version number
  - Customer impact

All communications will be acknowledged the next working day.



## Software Performance Report

Product:	Customer:
Version:	Company Name/Address:
Serial Number:	
Purchased From:	
	Phone Number:
	E-Mail Address:

Host:       Apollo       PC  
               DECstation       Sun-3  
               HP 9000 Series 300/400       Sun-4/SPARC  
               HP 9000 Series 700       VAX/VMS  
               IBM RS/6000       Other \_\_\_\_\_

Operating System \_\_\_\_\_ Version \_\_\_\_\_

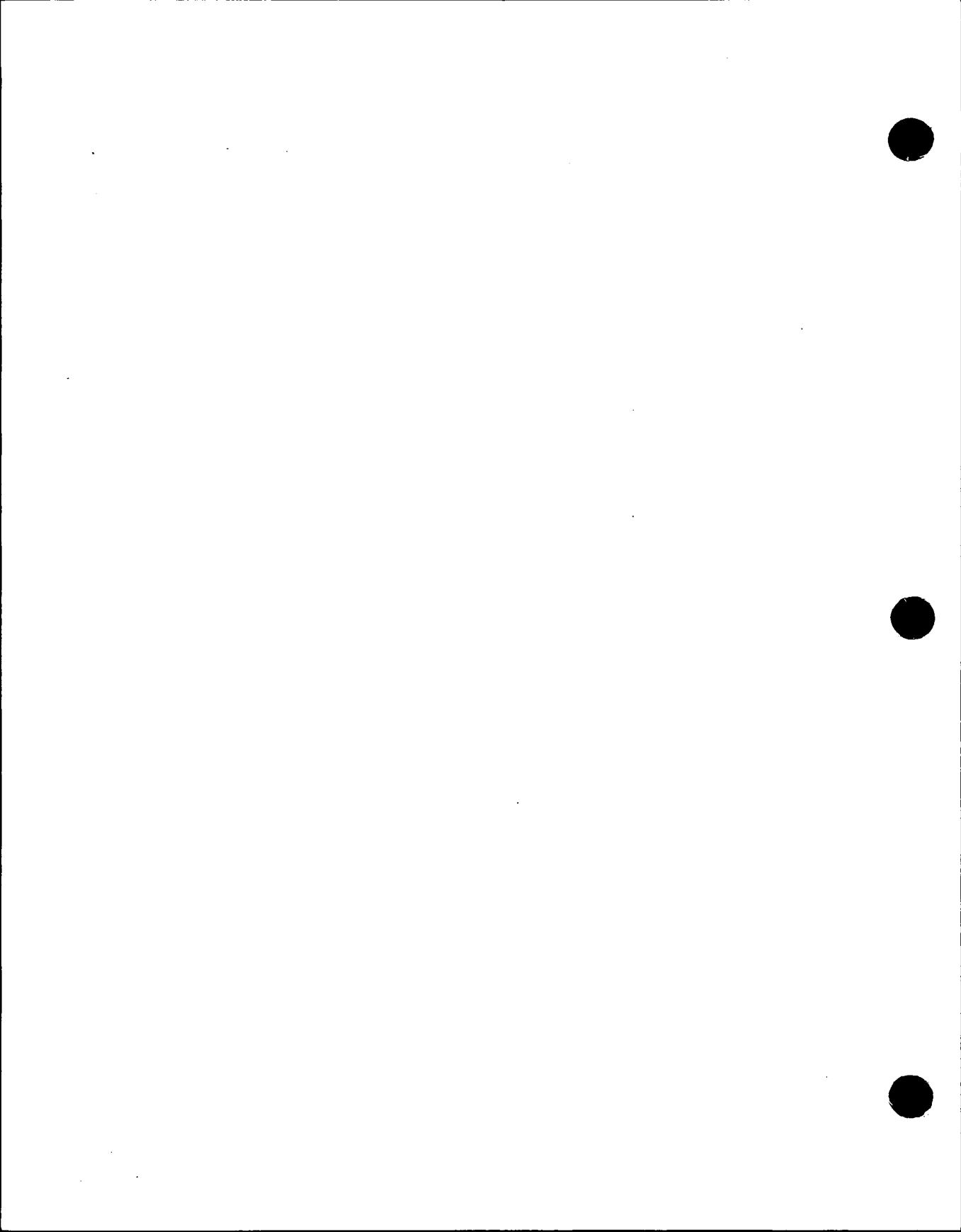
Report Type: Customer Impact:

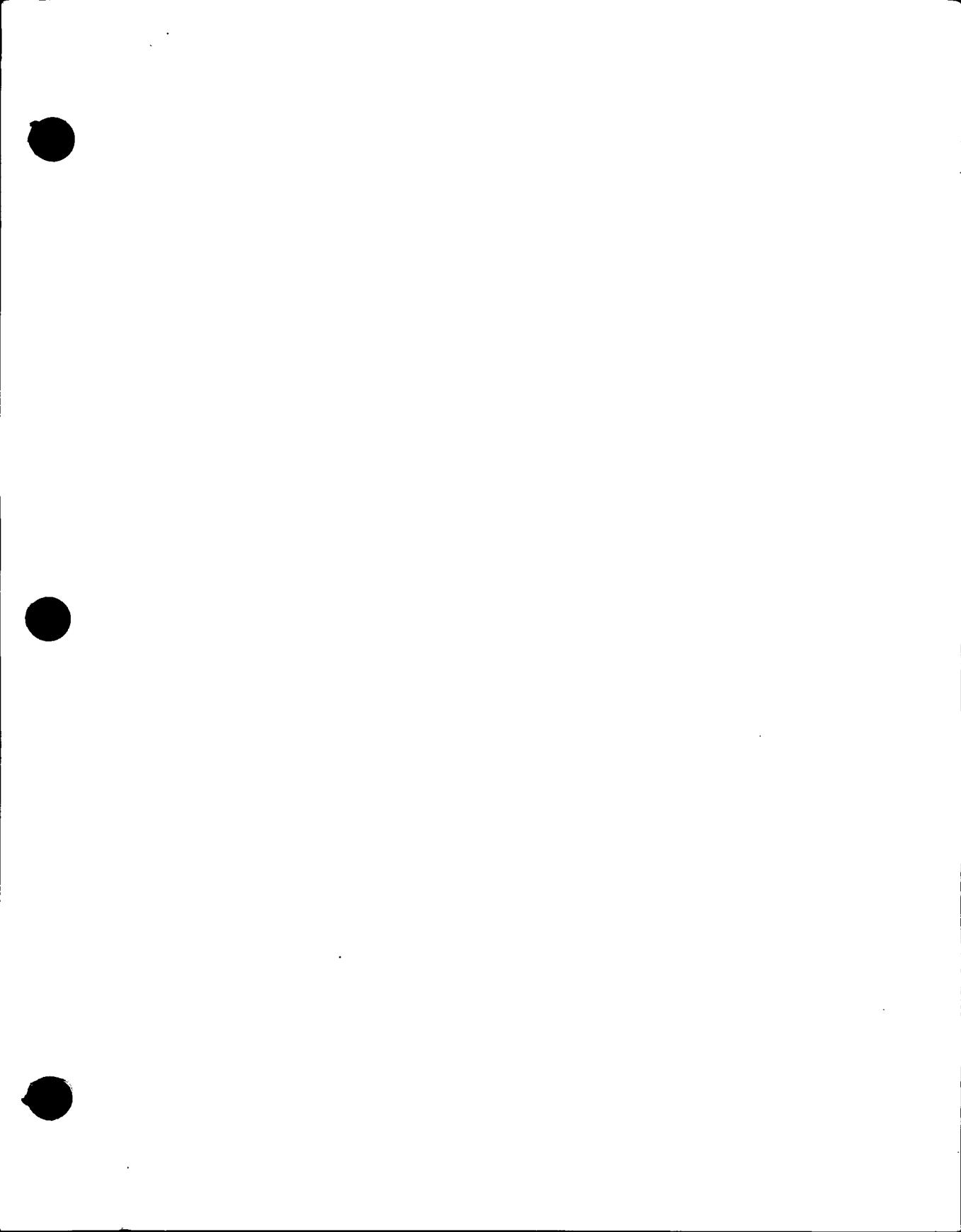
(Attach additional pages if necessary to complete the items below.)

Problem Description:

Commands or lines of code used to duplicate problem:

Workaround:







***Microtec  
Research***

**2350 Mission College Blvd.**

**Santa Clara, CA 95054**

**Tel. 408.980.1300**

**Toll Free 800.950.5554**

**FAX 408.982.8266**



*Microtec  
Research*

**ASM68K**  
*Installation  
Guide*

100008-009

## TRADEMARKS

Microtec® and Paragon® are registered trademarks of Microtec Research, Inc.

MRI ASMTM, MRI CTM, MRI FORTRAN™, MRI Pascal™, MRI XRAY™, Source Explorer™, XHM302™, XRAY™, XRAY Debugger™, XRAY In-Circuit Debugger™, XRAY In-Circuit Debugger Monitor™, XRAY MasterWorks™, XRAY/MTD™, XRAY180™, XRAY29K™, XRAY51™, XRAY68K™, XRAY80™, XRAY86™, XRAY88K™, XRAY960™, XRAYG32™, XRAYH83™, XRAYH85™, XRAYM77™, XRAYSP™, XRAYTX™, and XRAYZ80™ are trademarks of Microtec Research, Inc.

Other product names mentioned in this document are trademarks or registered trademarks of their respective companies.

## RESTRICTED RIGHTS LEGEND

If this product is acquired under the terms of a: *DoD contract*: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of 252.227-7013.

*Civilian agency contract*: Use, reproduction, or disclosure is subject to 52.227-19 (a) through (d) and restrictions set forth in the accompanying end user agreement. Unpublished rights reserved under the copyright laws of the United States.

MICROTEC RESEARCH, INCORPORATED  
2350 MISSION COLLEGE BLVD.  
SANTA CLARA, CA 95054

© Copyright 1988, 1989, 1990, 1991, 1992 Microtec Research, Inc. All rights reserved. No part of this publication may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical or otherwise, without prior written permission of Microtec Research, Inc.

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.		
-002	Updated from V6.3 to V 6.4 the installation procedures for VAX/VMS, VAX/ULTRIX, Sun UNIX, Apollo AEGIS & DOMAIN/IX and HP-UX.	6/88	L.C.
-003	Updated from V 6.4 to V 6.5 the installation procedures for VAX/VMS, VAX/ULTRIX, Sun UNIX, Apollo AEGIS & DOMAIN/IX, and HP-UX.  Added the IBM-PC/MS-DOS installation procedures.	8/89	L.C.
-004	Printed at 80% for new packaging.	11/89	L.C.
-005	For V 6.4, merged the IBM-PC/MS-DOS chapter into this guide.	12/89	L.C.
-006	Printed V 6.4 at 80% for new packaging.  Updated from V 6.5 to 6.6 the installation procedures for VAX/VMS, VAX/ULTRIX, Sun UNIX, Apollo AEGIS & DOMAIN/IX, HP-UX, and IBM-PC/MS-DOS.  Added the SONY NEWS installation procedures.	2/90	L.C.
-007	Added the SCO UNIX System V/386 installation procedures.	1/91	L.W.
-008	Removed flexed hosts.	9/91	L.L.
-009	Removed HP/HP-UX flexed host.	8/92	D.C.



The Microtec Research ASM68K assembler package consists of the following: the ASM68K Assembler, the LNK68K Linker, and the LIB68K Object Module Librarian.

## About This Guide

This guide describes how to install the Microtec Research MCC68K C Compiler for the Motorola 68000 family of microprocessors on the following host computers and operating systems:

- 80386 computers using SCO UNIX System V/386
- Digital Equipment Corporation VAX computers using VMS
- PC and compatible computers using MS-DOS or PC-DOS

If your host is supported by Flexible Licensing, refer to the Flexible License Manager documentation and ignore the installation instructions in this guide. The following hosts are supported by Flexible Licensing:

- Apollo computers using DOMAIN/OS
- Digital Equipment Corporation DECstation computers using ULTRIX
- Hewlett-Packard HP 9000 Series 300 / 400 computers using HP-UX
- Hewlett-Packard HP 9000 Series 700 computers using HP-UX
- IBM RS/6000 computers using AIX
- Motorola Delta Series Microcomputer using UNIX/System V
- Sun Microsystems Sun-3 computers using SunOS
- Sun Microsystems Sun-4 computers using SunOS

This guide contains the following chapters:

- Chapter 1, *PC / DOS Installation*, describes how to install the Microtec Research ASM68K assembler package on a PC or compatible using the MS-DOS or PC-DOS operating system.
- Chapter 2, *SCO UNIX System V/386 Installation*, describes how to install the Microtec Research ASM68J assembler package on 80386 computers using SCO UNIX System V/386.
- Chapter 3, *VAX / VMS Installation*, describes how to install the Microtec Research on Digital Equipment VAX computers using VMS.

## About the Documentation Set

The Microtec Research ASM68K package consists of the following: the ASM68K Assembler, LNK68K Linker, and LIB68K Object Module Librarian. The documentation set describing the Microtec Research ASM68K software consists of this *Installation Guide*, a *User's Guide*, and a *Reference Manual*.

- The *Microtec Research Installation Guide* contains host computer and operating system-specific instructions on how to install the ASM68K package.
- The *Microtec Research User's Guide* contains host computer and operating system-specific syntax for invoking the macro string preprocessor, assembler, linker, object module librarian, and conversion programs.
- The *Microtec Research Reference Manual* contains complete descriptions of the macro string preprocessor, assembler syntax and directives, linker commands, and librarian commands. Sample program listings are used as examples for the assembler, linker, and object module librarian. Error and warning messages are described.

## Related Publications

The *Microtec Research ASM68K User's Guide* is written for the experienced program developer and assumes the developer has a working knowledge of the 68000 family of microprocessors. Although it provides several useful and informative program examples, this documentation does not describe the microprocessor itself. For such information, refer to:

- *Motorola Programmer's Reference Manual*, M68000PM/AD REV1
- *The Motorola M68000 Resident Structured Assembler Reference Manual*, M68KMASM/D8
- *Motorola 68000 Microprocessor User's Manual*, M68000UM(AD4)
- *Motorola 68020 Microprocessor User's Manual*, MC68020UM/AD REV 1
- *Motorola MC68881/MC68882 Floating-Point Coprocessor User's Manual*, MC68881UM/AD REV 1
- *Motorola 68030 Enhanced 32-Bit Microprocessor User's Manual*, MC68030UM/AD
- *MC68040 Enhanced 32-Bit Third Generation Microprocessor User's Manual*, MC68040UM/AD
- *Motorola 68851 Paged Memory Management Unit User's Manual*, MC68851UM/AD
- *Motorola CPU Central Processor Unit Reference Manual*, CPU32RM/AD
- *Motorola M68000 Family Reference*, M68000FR/AD
- *Motorola M68040 Microprocessor User's Manual*, M68040UM/AD

Refer to the following publications for further information about the software development process and the C programming language:

- *C: A Reference Manual*, 3rd ed., by Samuel P. Harbison and Guy L. Steele Jr., Prentice-Hall, Inc., 1991.
- *The C Programming Language*, 2nd ed., Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1988.

Refer to the following publications for further information about Microtec Research tools:

- *MCC68K Documentation Set*, Microtec Research, Inc.

This documentation set describes how to use the MCC68K ANSI C Cross Compiler.

- *XRAY68K Documentation Set*, Microtec Research, Inc.

This documentation set describes how to use the XRAY68K Debugger.

- *Microtec Research Flexible License Manager Documentation Set*, Microtec Research, Inc.

This documentation set describes how to install and manage licenses for Microtec Research products that use Flexible Licensing.

## Notational Conventions

This guide uses the conventions listed in Table P-1 (unless otherwise noted).

**Table P-1. Notational Conventions**

Symbol	Name	Usage
{ }	Curly Braces	Encloses a list from which you must choose an item.
[ ]	Square Brackets	Encloses items that are optional.
...	Ellipsis	Indicates that you may repeat the preceding item zero or more times.
	Vertical Bar	Separates items in a list.
	Punctuation	Punctuation other than that described here must be entered as shown.
	Typewriter font	Represents user input in interactive examples.
	<i>Italics</i>	Indicates a descriptive item that should be replaced with an actual item.

## Questions and Suggestions

To help us respond to any questions or suggestions regarding this product, we have provided two response forms in the back of this guide.

The first form is a Reader's Response sheet, which is used to help correct and improve our documentation. We always appreciate your comments, and by completing this form, you can participate directly in the revisions of future publications. The Reader's Response sheet is directed to Technical Publications.

The second form is a Software Performance Report which should be completed if you encounter any errors or problems with Microtec Research software. This report is directed to Technical Support.



# Contents

---

## Preface

About This Guide .....	v
About the Documentation Set .....	vi
Related Publications .....	vii
Notational Conventions .....	viii
Questions and Suggestions .....	viii

## 1 PC / DOS Installation

Overview .....	1-1
Distribution Format .....	1-1
Installation Procedure .....	1-1

## 2 SCO UNIX System V/386 Installation

Distribution Format .....	2-1
Installation Procedure .....	2-1
Getting Started .....	2-2

## 3 VAX / VMS Installation

Distribution Format .....	3-1
Installation Procedure .....	3-1
Getting Started .....	3-3

# Tables

---

Table P-1. Notational Conventions .....	viii
Table 3-1. Device Types .....	3-2



# PC / DOS Installation 1

---

## Overview

This chapter describes how to install the Microtec Research ASM68K Assembler, Linker, and Librarian on an IBM-PC or compatible computer using the MS-DOS or the PC-DOS operating system. For specific system requirements, refer to your release notes.

The operating system prompt used in the directions is C>. Your system may be configured with a different prompt.

## Distribution Format

The ASM68K assembler package for DOS is distributed on diskettes. The license is for a single computer; copies can be made for backup purposes only.

### Note

Before installation, make backup copies of the diskettes using the DOS DISKCOPY command.

Since the distribution programs do not fit on a single diskette, it is inconvenient to execute them on systems without a hard disk. This guide assumes that your PC has a hard disk with 2MB available.

## Installation Procedure

To install the ASM68K files, complete the following steps:

1. Boot DOS.
2. Insert the diskette labeled 1 of *n* into drive A.

3. Enter:

```
C> a:install
```

This command creates the distribution directory and copies the distribution files into it. Instructions will appear on your display screen. You will be prompted for:

- The target hard drive (default: C)
- The distribution directory (default: \ASM68K)

You can enter **CTRL-C** (press both keys simultaneously) at any prompt to abort the installation.

**Warning**

If the distribution directory already exists, the installation program displays a message warning you that all the files in the distribution directory will be erased. You can select a different distribution directory at this time.

For information on the files provided with this distribution, refer to your *Release Notes*.

4. Insert the other diskettes into drive A as prompted.
5. Edit your search path in the **AUTOEXEC.BAT** file to include the distribution directory. For example, if your current path is C: \bin;\dir1, you would append \ASM68K to your current path by entering:

```
PATH C:\bin;\dir1;\ASM68K
```

You must reboot your system (see the next step before rebooting) after modifying the **AUTOEXEC.BAT** file so that DOS will automatically use your new values or change your current path setting for this session by entering:

```
C> PATH C:\bin;\dir1;\ASM68K
```

6. Edit/create the **CONFIG.SYS** file to contain the lines:

```
FILES = 11  
BUFFERS = 11
```

**Note**

These are the minimum values required; other applications may require larger values. Increase these values if many include files are used. Care should be exercised when increasing these values because file and buffer parameters reduce available memory.

Always reboot your system after modifying the **CONFIG.SYS** file so that DOS will use your new values.

You have now completed the ASM68K installation.



# SCO UNIX System V/386 Installation 2

---

This chapter describes how to install the Microtec Research ASM68K Assembler, Linker, and Librarian on a standard 386-based personal computer using the SCO UNIX System V/386 operating system.

The operating system prompt used in the directions is a percent sign (%). Your system may be configured with a different prompt.

## Distribution Format

The ASM68K assembler package for the SCO UNIX System V operating system is distributed on 5.25 inch (360K or 1.2MB) or 3.5 inch (720K or 1.4MB) floppy diskettes in a UNIX tar format that is readable by the SCO UNIX custom installation program.

## Installation Procedure

To install the ASM68K files, complete the following steps:

1. Log in to an account that has super-user privileges. Since ASM68K files will be copied to directories requiring root write permission, you must have super-user privileges to complete the installation.
2. Run the **custom** installation program by entering:

```
% custom -m /dev/xxxx
```

where xxxx is the device name. Refer to the system file */etc/default/tar* for your device name.

For more information on **custom**, a menu-driven program that is resident on your SCO UNIX system, refer to the *SCO UNIX System Administrator's Reference Manual*.

3. Add */usr/mri/bin* to your path variable if it is not already there. If you are using the C shell (**csh**), enter:

```
% set path=($path /usr/mri/bin)
```

If you are using the Bourne shell (**sh**), enter:

```
% PATH="$PATH:/usr/mri/bin"
```

4. Change your .login (csh) or .profile (sh) file to include /usr/mri/bin in your path definition.
5. If you are using the C shell (csh), you must make executable files available from the current directory by entering:

```
% rehash
```

You have now completed the ASM68K installation.

## Getting Started

After installation, see the *Microtec Research ASM68K User's Guide* for information on how to invoke ASM68K.

# VAX / VMS Installation 3

---

This chapter describes how to install the Microtec Research ASM68K Assembler, Linker, and Librarian on a VAX host computer using the VMS operating system.

The operating system prompt used in the directions is a dollar sign (\$). Your system may be configured with a different prompt.

## Distribution Format

The Microtec Research ASM68K package for VAX/VMS is available on:

- 9-track magnetic tape (1600 bpi) in **BACKUP** format
- TK50 cartridge tape in **BACKUP** format

## Installation Procedure

The ASM68K assembler package for VMS is distributed on either magnetic or cartridge tape. The first step in installing the programs is to copy the files from the tape onto the host computer system's working disk. Once the files are copied, you should execute the installation command procedure to complete the installation.

To install the ASM68K assembler package, follow the listed steps:

1. Log in to the system installation account, **SYSTEM**. You must have the privileges of a system manager in order to log into this account.
2. Set the default directory to **SYS\$UPDATE** by entering:

```
$ SET DEFAULT SYS$UPDATE:
```

The **SYS\$UPDATE** directory is used during the installation procedure as a temporary directory. Files are read into the **SYS\$UPDATE** directory and then moved to other directories. All files created in this directory are automatically removed after the installation process.

3. Physically mount the distribution tape on an appropriate drive. Verify that the tape is write-protected by setting the write protection switch in the **safe** position (cartridge tape) or by removing the write-enable ring if one is on the tape (9-track tape).

4. If the tape drive on your system has a density switch on it, set it for the density indicated on the distribution tape.

The name of the tape drive can vary from one computer system to another. See your system administrator for the tape device name that is correct for you. Table 3-1 shows some common devices and their corresponding mnemonic names.

**Table 3-1. Device Types**

Device Type	Mnemonic
TU78	MF $c$ n:
TE16, TU16, TU77	MT $c$ n:
TS11, TSV05, TU80	MS $c$ n:
TK50	MU $c$ n:

In the mnemonics shown,  $c$  refers to the controller (usually A), and  $n$  refers to the unit number of the drive (usually 0).

**Note**

These instructions use *MTA0* as a sample device name. Use the appropriate tape drive designation for your system in place of *MTA0*.

5. Logically mount the tape by entering:

\$ ALLOCATE *MTA0*:

\$ MOUNT/FOREIGN *MTA0*:

6. Create an empty directory and move to it, or change your current directory to an existing empty directory. This directory is your distribution directory.

\$ CREATE/DIRECTORY [*.distribution\_directory*]

\$ SET DEFAULT [*.distribution\_directory*]

7. Copy the files from the tape to your *distribution\_directory* by entering:

\$ BACKUP *MTA0::.\*.\*.\**;

**Note**

Files with duplicate file names will be overwritten. Refer to your *Release Notes* for information on the files provided with this distribution and the default installation directories..

8. Dismount the tape by entering:

```
$ DISMOUNT MTA0;
```

```
$ DEALLOCATE MTA0;
```

9. Run the installation procedure command file by entering:

```
$ @MRIINSTAL
```

10. Log out and log back in to activate the CLI tables, which are updated by the installation command file. Other users logged in during this installation must also log out and log back in before using the executable program images included on the ASM68K distribution.

You have now completed the ASM68K installation.

## Getting Started

After installation, see the *Microtec Research ASM68K User's Guide* for information on how to invoke ASM68K.



# Index

---

## Symbols

9-track tape  
VAX/VMS 3-1

## A

AUTOEXEC.BAT file 1-2

## C

Cartridge tape  
VAX/VMS 3-1  
CONFIG.SYS file 1-3  
Conventions, notational viii

## D

Diskette  
SCO UNIX 2-1  
Distribution format  
PC/DOS 1-1  
Distribution media, copying  
PC/DOS 1-2  
Documentation, description vi  
DOS installation 1-1 to 1-3  
disk distribution 1-1  
distribution format 1-1  
distribution media  
diskette 1-1  
installation procedure 1-1  
system configuration, modifying 1-3

## F

Floppy diskette  
SCO UNIX 2-1

## I

IBM-PC  
(see DOS installation)

## Installation

DOS 1-1 to 1-3  
SCO UNIX 2-1 to 2-2  
VAX/VMS 3-1 to 3-3

## Introduction

vi

## M

MS-DOS  
(see DOS installation)

## N

Notational conventions viii

## O

Operating systems supported v

## P

PC-DOS  
(see DOS installation)

## S

SCO UNIX installation 2-1 to 2-2  
distribution format 2-1  
getting started 2-2  
installation procedure 2-1

Supported operating systems v

System configuration, modifying  
PC/DOS 1-3

System V/386  
(see SCO UNIX installation)

## U

UNIX  
SCO UNIX 2-1

**V**

**VAX/VMS installation** 3-1 to 3-3

    distribution format 3-1

    getting started 3-3

    installation procedure 3-1

**VMS**

    (see **VAX/VMS installation**)

## Reader's Response

Please complete and return the following Reader's Response form to help us improve our documentation. Your comments and suggestions are always appreciated.

Product Name \_\_\_\_\_

Your Name \_\_\_\_\_

Company \_\_\_\_\_

Phone Number \_\_\_\_\_

Manual Date/Version \_\_\_\_\_

Title \_\_\_\_\_

Address \_\_\_\_\_

What is your level of programming experience?

Advanced

Intermediate

Beginner

In this language?

Advanced

Intermediate

Beginner

Circle the number which best expresses your rating of the information in this manual (*5 is the highest rating*):

Complete?

1

2

3

4

5

Easy to understand?

1

2

3

4

5

Accurate?

1

2

3

4

5

Helpful examples?

1

2

3

4

5

Well organized?

1

2

3

4

5

Helpful procedures?

1

2

3

4

5

Easy to Find?

1

2

3

4

5

At an appropriate level?

1

2

3

4

5

Easy to Read?

1

2

3

4

5

Overall rating?

1

2

3

4

5

3.) Which programming language do you use?

---

4.) Does the manual provide all the necessary information? If not, what is missing?

---

5.) Are more examples needed? If so, where?

---

6.) Did you have any difficulty understanding descriptions or wording? If so, where?

---

7.) Which sections of this manual are the most important? The most helpful?

---

8.) What are the best features of this manual?

---

9.) What are the areas in this manual that need improvement?

---

10.) Did you find any errors in this manual? (Specify section and page number.)

---



## **Software Performance Report Instructions**

If you encounter problems or errors when using Microtec Research software products (including documentation), you can send a Software Performance Report (SPR):

- Electronically through e-mail to **support@mri.com**

or

- By completing the SPR form on the next page and returning it to:

**Microtec Research  
Software Performance Reports  
2350 Mission College Boulevard  
Santa Clara, CA 95054**

All SPRs are directed to our Technical Support Department for analysis and response.

To make it easier for us to respond quickly to your problem, please include the following:

1. Give a complete description of the problem, error, or suggestion, including the exact wording of any error messages.
2. If possible, isolate the problem by providing a small example.
3. If you are mailing or FAXing an SPR and your error example is longer than one page of source code, please provide all information on a tape or floppy disk.
4. Include command files, listings, load maps, include files, data files, and all other relevant material with the message.
5. If applicable, send sample input and output listings.
6. If the output is from a compiler, send a mixed (source/assembly code) listing.
7. Be sure to include the following information:
  - Product name and version number
  - Serial number
  - Name of contact
  - Company purchased from
  - Company name and a shipping address
  - Phone number and/or FAX number
  - Host machine
  - Host operating system and version number
  - Customer impact

All communications will be acknowledged the next working day.



## Software Performance Report

Product:	Customer:
Version:	Company Name/Address:
Serial Number:	
Purchased From:	Phone Number:
	E-Mail Address:

Host:       Apollo       PC  
               DECstation       Sun-3  
               HP 9000 Series 300/400       Sun-4/SPARC  
               HP 9000 Series 700       VAX/VMS  
               IBM RS/6000       Other \_\_\_\_\_

Operating System \_\_\_\_\_ Version \_\_\_\_\_

Report Type: Customer Impact:

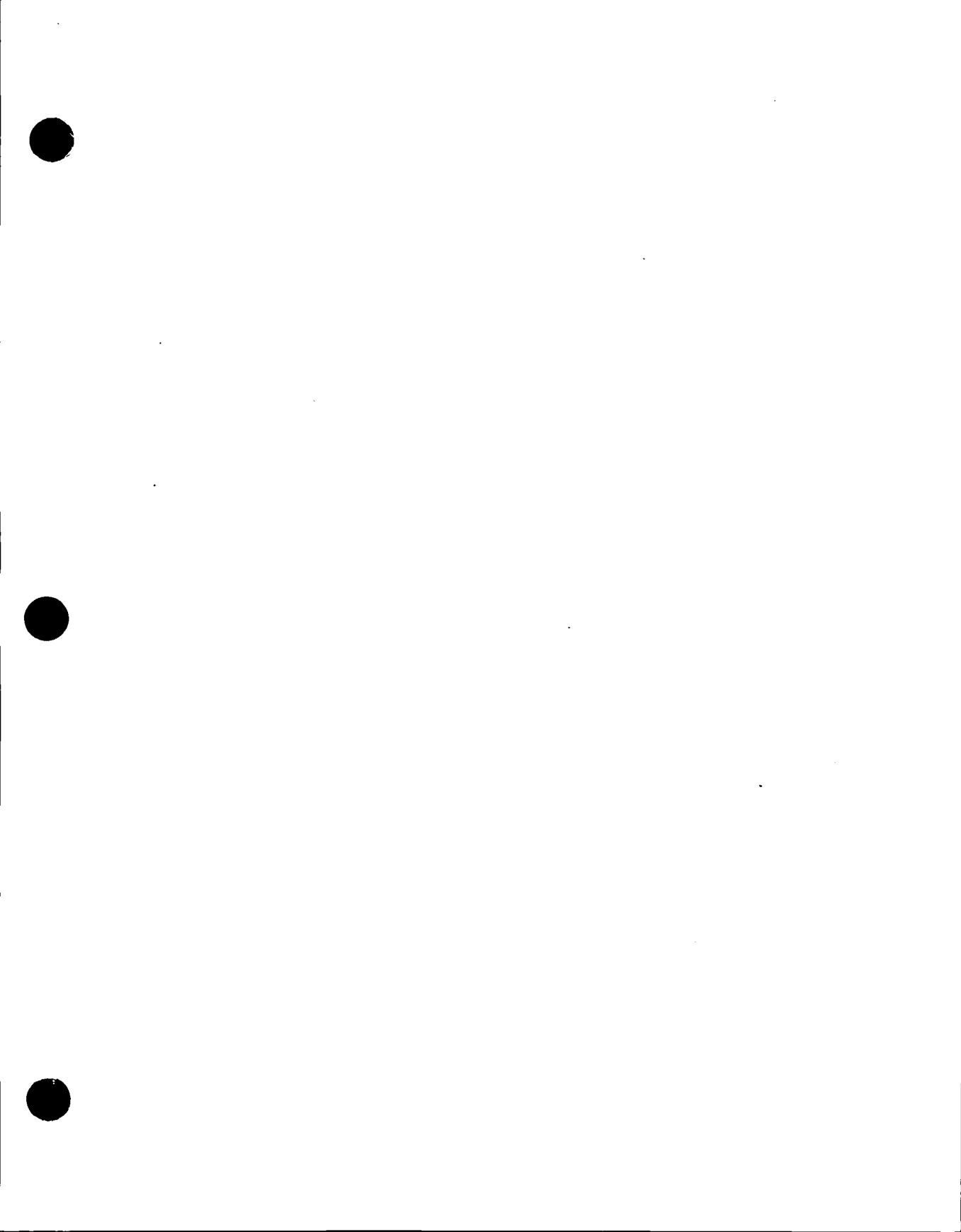
(Attach additional pages if necessary to complete the items below.)

Problem Description:

Commands or lines of code used to duplicate problem:

Workaround:







***Microtec  
Research***

2350 Mission College Blvd.

Santa Clara, CA 95054

Tel. 408.980.1300

Toll Free 800.950.5554

FAX 408.982.8266



## **Release Notes**

**Microtec Research ASM68K**

**Assembler, Linker, and Librarian**

**Release 6.9C**

100673-022

## **TRADEMARKS**

Microtec®, the Microtec logo, and XRAY® are registered trademarks of Microtec Research, Inc.

Source Explorer™, XRAY In-Circuit Debugger™, XRAY In-Circuit Debugger Monitor™, and XRAY MasterWorks™ are trademarks of Microtec Research, Inc.

Other product names mentioned in this document are trademarks or registered trademarks of their respective companies.

## **RESTRICTED RIGHTS LEGEND**

If this product is acquired under the terms of a: *DoD contract*: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of 252.227-7013. *Civilian agency contract*: Use, reproduction, or disclosure is subject to 52.227-19 (a) through (d) and restrictions set forth in the accompanying end user agreement. Unpublished rights reserved under the copyright laws of the United States.

MICROTEC RESEARCH, INCORPORATED  
2350 MISSION COLLEGE BLVD.  
SANTA CLARA, CA 95054

Copyright © 1987, 1988, 1989, 1990, 1991, 1992, 1993 Microtec Research, Inc. All rights reserved. No part of this publication may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical or otherwise, without prior written permission of Microtec Research, Inc.

## *Table of Contents*

<b>Installation .....</b>	1
<b>UNIX Hosts .....</b>	1
System Requirements .....	1
Installation Instructions .....	1
Motorola Delta88K Installation Procedure .....	2
Files on the Distribution .....	3
<b>PC Hosts .....</b>	3
System Requirements .....	3
Installation Instructions .....	3
DOS Memory Management System .....	3
Files on the Distribution .....	4
<b>VMS Hosts .....</b>	5
System Requirements .....	5
Installation Instructions .....	5
Files on the Distribution .....	5
<b>Using 68K tools with XRAY .....</b>	6
Patch68k Utility .....	6
<b>ASM68K Assembler .....</b>	7
Operational Changes .....	7
Enhancements .....	8
Corrections .....	9
Known Problems .....	10
<b>LNK68K Linker .....</b>	12
Operational Changes .....	12
Enhancements .....	13
Corrections .....	14
Known Problems .....	15
<b>LIB68K Librarian .....</b>	15
Operational Changes .....	15
Enhancements .....	16
Corrections .....	16

<b>Known Problems .....</b>	<b>16</b>
<b><i>Documentation Errata .....</i></b>	<b>16</b>

This document describes the operational changes, enhancements, problem corrections, and known problems for the ASM68K assembler package.

Please fill out your warranty registration card and return it to Microtec Research, Inc. Thank you.

## ***Installation***

### **UNIX Hosts**

#### **System Requirements**

ASM68K requires one of the following configurations:

- Apollo Domain workstation running DOMAIN/OS version 10.3 or newer
- DECstation/DECsystem running ULTRIX version 4.2 or newer
- HP 9000 Series 300/400 running HP-UX 8.0 or newer
- HP 9000 Series 400 running DOMAIN/OS version 10.3 or newer
- HP 9000 Series 700 running HP-UX 8.0 or newer
- IBM RS/6000 system running AIX version 3.2 or newer
- Motorola Delta88K system running System V/88 release 4 version 3 or newer
- Sun-3 workstation running SunOS version 4.0.3 or newer
- SPARCstation running SunOS version 4.1.1 or newer

#### **Installation Instructions**

This product is installed using Flexible Licensing, Version 2, on all UNIX hosts (except the Motorola Delta88K system). Installation instructions appear in the *Flexible License Manager Documentation Set* (document number 100675), which contains an *Installation Guide* and *User's Guide*. Also, read the *Flexible License Manager Release Notes* (document number 100953) for important, recent information.

## Motorola Delta88K Installation Procedure

To install the ASM68K files, complete the following steps:

1. Log in to an account that has super-user privileges. Since ASM68K files will be copied to protected directories, you must have super-user privilege to complete the installation.
2. Physically mount the distribution media on an appropriate drive with the write protection switch in the **safe** position.
3. Create the installation directory (usually **/usr/mri**), if it does not exist, and move to it:

```
% mkdir Install_dir
% cd Install_dir
```

4. Read the media by entering:

```
% tar -xvfp /dev/rmt/ctape1
```

where **/dev/rmt/ctape1** is the device name of your cartridge tape drive. The names of files are displayed on your terminal as they are read into **Install\_dir**.

5. Add the directory (say **/usr/mri/bin**) containing the executable files to your search path as shown below.

Using C shell (**csh**):

```
% set path=($path /usr/mri/bin)
```

Using Bourne shell (**sh**):

```
% PATH="$PATH:/usr/mri/bin"
% export PATH
```

6. Edit your **.login** (**csh**) or **.profile** (**sh**) file to add the ASM68K directory to your search path.
7. If you are using the C shell (**csh**), you must make executable files available from the current directory by entering:

```
% rehash
```

You have now completed the ASM68K installation.

## Files on the Distribution

A complete list of files appears in the file:

*install\_dir/asm68k/asm68k.fil*

where *install\_dir* is normally */usr/mri*. The files are installed in the following directories:

Toolkit Executables	<i>install_dir/bin</i>
Support and Test Files	<i>install_dir/asm68k</i>
MasterWorks Files	<i>install_dir/master</i> (Sun-4 Only)

## PC Hosts

### System Requirements

ASM68K now requires at least an AT class computer with a minimum of one megabyte of extended memory available above the 1 megabyte address space. The operating systems supported include the following:

- MS-DOS version 3.2 or newer
- PC-DOS version 3.2 or newer
- SCO UNIX System V.3.2 or newer

### Installation Instructions

For installation information, refer to your *Installation Guide*.

### DOS Memory Management System

Microtec Research products use the Rational Memory Management System on PC host systems. For information on this memory manager and a list of messages it produces, refer to the *DOS Memory Management Supplement*.

You can modify your memory configuration using the *product.VMC* file (where *product* is the name of your product).

### Example:

```
MCC68K.VMC  
ASM68K.VMC  
XHS68K.VMC
```

Use any ASCII text editor to create this file and enter the appropriate parameters. The parameters are described in the following section and in the *DOS Memory Management Supplement* (note that the section *Setting Memory Parameters for XRAY* in this supplement actually applies to all Microtec Research products).

All products search for the *product.VMC* file in the current directory. XRAY also searches for this file using the path(s) specified by the **XRAYLIB** environment variable.

### Using Expanded Memory

Microtec Research products will operate with expanded memory emulators such as 386<sup>MAX</sup>, QEMM, and EMM386. Emulators should be set to provide a minimum of 1 megabyte (4 megabytes for XRAY) of expanded memory so that operations requiring a significant amount of memory run as fast as possible. For example, loading a big file into XRAY requires a large amount of memory for the symbol table. Performance improves with larger memory settings.

Microtec Research products will not operate with expanded memory standard LIMS EMS.

Microtec Research products recognize extended memory by default. In order to recognize expanded memory, the following entry must be present in your **ASM68K.VMC** file:

**EMS=n**

where *n* specifies the number of kilobytes of EMS memory used for swapping before swapping to disk. The default value of this parameter is 0. Swapping automatically spills from EMS to disk if there is not enough EMS memory.

### Files on the Distribution

A complete list of files appears in the file:

*install\_device:\asm68k\asm68k.fil*

where *install\_device* is normally the C drive. The files are installed in the following directory:

*install\_device:\asm68k*

## VMS Hosts

### System Requirements

ASM68K requires one of the following configurations:

- MicroVAX system running MicroVMS version 5.3 or newer
- VAX workstation running VMS version 5.3 or newer

### Installation Instructions

For installation information, refer to your *Installation Guide*.

### Files on the Distribution

A complete list of files appears in the file **asm68k.fil** in the support files directory.  
The files are installed in the following directories:

Toolkit Executables      **SYS\$COMMON:[SYSEXEC]**

Support and Test Files  
**SYS\$COMMON:[SYSHLP.EXAMPLES.MRI68K]**

# Using 68K tools with XRAY

## Patch68k Utility

This utility is used to set the internal IEEE chip type. XRAY68K versions prior to version 2.3 will not accept all of the chip types set by the 4.3 MCC68K development tools. For example, the most current version of MCC68K uses CPU32 to represent the 68332 family of processors. If a current version of XRAY68K that accepts CPU32 is not available, it may be necessary to patch the chip type with 68332 using this utility.

Here is the command line usage for patch68k.

```
patch68k <legal_chip_type> <ieee_file>
```

Where *<legal\_chip\_type>* is one of the following:

```
[ 68000 | 68302 | 68010 | CPU32 | 68331 |  
 68332 | 68333 | 68020 | 68030 | 68040 ]
```

On VAX/VMS, the patch68k utility exists as a foreign image and the following DCL command will allow this executable to be invoked.

```
patch68k := $ sys$common:[sysexe]patch68k.exe
```

Once this is done, the patch68k utility will behave as described above.

# **ASM68K Assembler**

## **Operational Changes**

### **TDIVS and TDIVU Instructions**

The **TDIVS** and **TDIVU** instructions now generate a warning that they will be discontinued in the next release. Equivalent instructions **DIVSL** and **DIVUL** should be used instead.

### **XDEF vs. XREF Behavior**

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. For example, the following is now legal:

```
XREF sym1  
XDEF sym1  
sym1:
```

### **CHIP Directive**

The **CHIP** directive no longer supports an absolute expression. Processor types are now processed as strings. i.e. 68EC030.

## Enhancements

### New Processor Support

Several new processor types are now supported by the assembler. Not all of the new processor types invoke unique behavior within the assembler. The internal values given below are only used to determine which instruction set will be allowed. The input processor string will be used when the processor type is output to the object file. The following table shows the behavior of all legal processor types:

Input Processor String	Instruction Set Allowed
68000	68000
68EC000	68000
68HC000	68000
68HC001	68000
68302	68000
68008	68000
68010	68010
68330	CPU32
68331	CPU32
68332	CPU32
68333	CPU32
68340	CPU32
CPU32	CPU32
68020	68020
68EC020	68020
68030	68030
68EC030	68EC030
68040	68040
68EC040	68EC040

## **Relocatable Expressions**

The divide (/) and multiply (\*) operators are now accepted in relocatable expressions. Any fractional results are truncated to integer values.

## **EQU Directive**

The EQU directive now supports simple forward references. A single symbol is accepted and no operators are allowed.

## **Macro Nesting Level**

(PC Only) Increased maximum macro nesting level from 8 to 20.

## **Corrections**

### **CHIP Directive**

The CPU32 processor type was not being recognized when specified in the CHIP directive. This caused the MOVEC instruction to not be recognized.

### **IRP Directive**

An IRP directive with no parameters is supposed to expand once. In the previous version, multiple expansions would randomly occur.

### **OPT BRL,BRS,OLD**

The behavior of OPT BRL,BRS, and OLD with a processor type of 68020 or greater was not correct.

### **FMOVE.M.X Instruction**

The FMOVE.M.X instruction was encoded incorrectly.

### **OPT NOBRL**

The BRL option can now be negated. This will have the effect of setting the relative branch size back to the default of BRW.

### **"-I" Include Path Option**

The "-I" include path option was not working with relative paths.

## Packed Decimal Floating Point

Packed decimal floating point values were not encoded correctly.

## Include File Line Number Limit

Include file line numbers had a limit of 1000 lines. When exceeded the line number would wrap around to 1. This limit has been increased to 10000.

## More Than 200 Absolute Sections

The assembler no longer core dumps when more than 200 absolute section are encountered.

## BTST Instruction

The assembler was allowing an illegal combination of operands for the BTST instruction.

## Known Problems

### Include File Nesting

Although the include file nesting level can be greater than four, the listing line numbers can display no greater than four levels.

### Negation of Floating-Point Numbers

A negation of a negative floating-point number will not produce a positive number. The following example illustrates the defect.

```
AA    FEQU    -1.0
00000008 F23C 4423 BF80      FMUL.S #-AA,FPO
                                0000
```

### DC Directive

For the **DC** assembler directive, the evaluation of the Assembly Program Counter (denoted by the "\*" character) is different from Motorola's assembler. The following example illustrates:

#### MRI

```
00000000 1234 0002      R  DC.W  $1234,*
```

#### MOTOROLA

```
00000000 1234 0000      R  DC.W  $1234,*
```

A future release of ASM68K will provide a flag for Motorola-compatibility. One can use "/\*-2" as a work-around if Motorola-compatibility is desired.

### XREF Symbols Used as Displacements

The assembler currently does not allow an external reference (**XREF**) to take the place of a displacement in addressing modes like

```
bd(An,Xi) or bd(PC,Xi)
```

This limitation exists in 68000-mode only.

### SECT Directive

The following **SECT** directive will not cause section '1' to be created, unlike Motorola's assembler. Instead, section X is created.

```
X  EQU    1  
SECT   X
```

# **LNK68K Linker**

## **Operational Changes**

### **LISTABS Default for S-records**

The default for **LISTABS INTERNALS** is now different depending on the output format. The default for IEEE format is **INTERNAL**s as before. The default for S-records output is now **NOINTERNAL**s.

### **SECTSIZE and ORDER Commands**

The **SECTSIZE** and **ORDER** commands no longer create a section if it does not exist. The **SECT** or **COMMON** command can be used to create a section from the command file. This new behavior eliminates the problem of getting "Section Mismatch" errors when the default section type for a section created by the **SECTSIZE** or **ORDER** command does not match the section type from an object, **SECT** command, or **COMMON** command.

### **PUBLIC Command**

A "Duplicate Public" error will no longer be generated when a **PUBLIC** command is used to override a library definition of the same symbol name.

The **PUBLIC** command will now override a library definition of the same symbol name and prevent the library module from being loaded if there are no other symbols needed from the library module.

### **Command File Comment Char '\*'\***

A warning is no longer generated if a '\*' command file comment is not the first white space character. An error will now be generated.

### **Chip Inconsistency**

Error "(314) Chip Inconsistency" has been down graded to a warning. Also, CPU32/68020 is no longer an inconsistent chip combination.

## Enhancements

### New Processor Support

Several new processors are now supported by the linker. Not all of the new processor types invoke unique behavior within the linker. The internal values shown below are only used when checking processor compatibility. The input processor string will be used to determine which processor type is output to the absolute file. The following table shows the behavior of all legal processor types:

Input Processor String	Internal Type For Checking Processor Compatibility
68000	68000
68EC000	68000
68HC000	68000
68HC001	68000
68302	68000
68008	68008
68010	68000
68330	CPU32
68331	CPU32
68332	CPU32
68333	CPU32
68340	CPU32
CPU32	CPU32
68020	68020
68EC020	68020
68030	68030
68EC030	68030
68040	68040
68EC040	68040

## **Linker Performance**

The symbol table manager has been enhanced. This results in a performance increase of 30 percent on large links.

## **START Command**

The START command will now take a symbol or value as an argument.

## **Reading MS-DOS Format Command Files**

The UNIX versions of the linker will now read MS-DOS format command files.

## **Corrections**

### **NOCASE Command**

The NOCASE command was not being accepted as a legal command.

### **LISTABS NOPUBLICS**

The LISTABS NOPUBLICS command was not eliminating the public symbol table from the absolute file.

### **INITDATA Problem**

If the initialized data in the first section put into the ??INITDATA section did not start at offset 0 from the start of the section, it would not be adjusted properly when output to the absolute file.

### **Register Aliveness**

The offset for the ?alive debug directive was not processed correctly.

### **Record Length in HP OMF Absolute Files**

The maximum record length for HP format is now 256 bytes. The previous setting of 512 bytes was incorrect.

### **Greater Than 100 consecutive LOAD Commands**

A Segmentation Fault was generated when there were more than 100 consecutive LOAD commands in a command file.

### **LOAD\_SYMBOLS Command**

The VMISC debug directive would sometimes cause a segmentation violation while processing a *load\_symbols* command.

### **IEEE ATM record**

The linker no longer places a spurious 0-byte at the end of ATM records.

### **Cannot Seek to Beginning of File**

The error "Cannot seek to beginning of file: <filename>" was reporting the wrong <filename>. This has been corrected. The proper <filename> is now reported in the error message.

### **Local Symbol Addresses in S-record Files**

Local symbol addresses in S-record files were incorrect. The section base was not being added in.

## **Known Problems**

### **-C Option**

The linker option *-Clinker\_command* will not support **INCLUDE file\_name** as the *linker\_command*.

## **LIB68K Librarian**

### **Operational Changes**

#### **Serial Number Display in Banner**

The librarian is no longer a serialized product. The serial number string has been removed from the code and banner.

## Enhancements

None.

## Corrections

None.

## Known Problems

### EXTRACT Command

The EXTRACT librarian command copies a library module to a file outside the library. It must be preceded by an OPEN or a CREATE command. If an error occurs prior to the EXTRACT command, the EXTRACT is not inhibited.

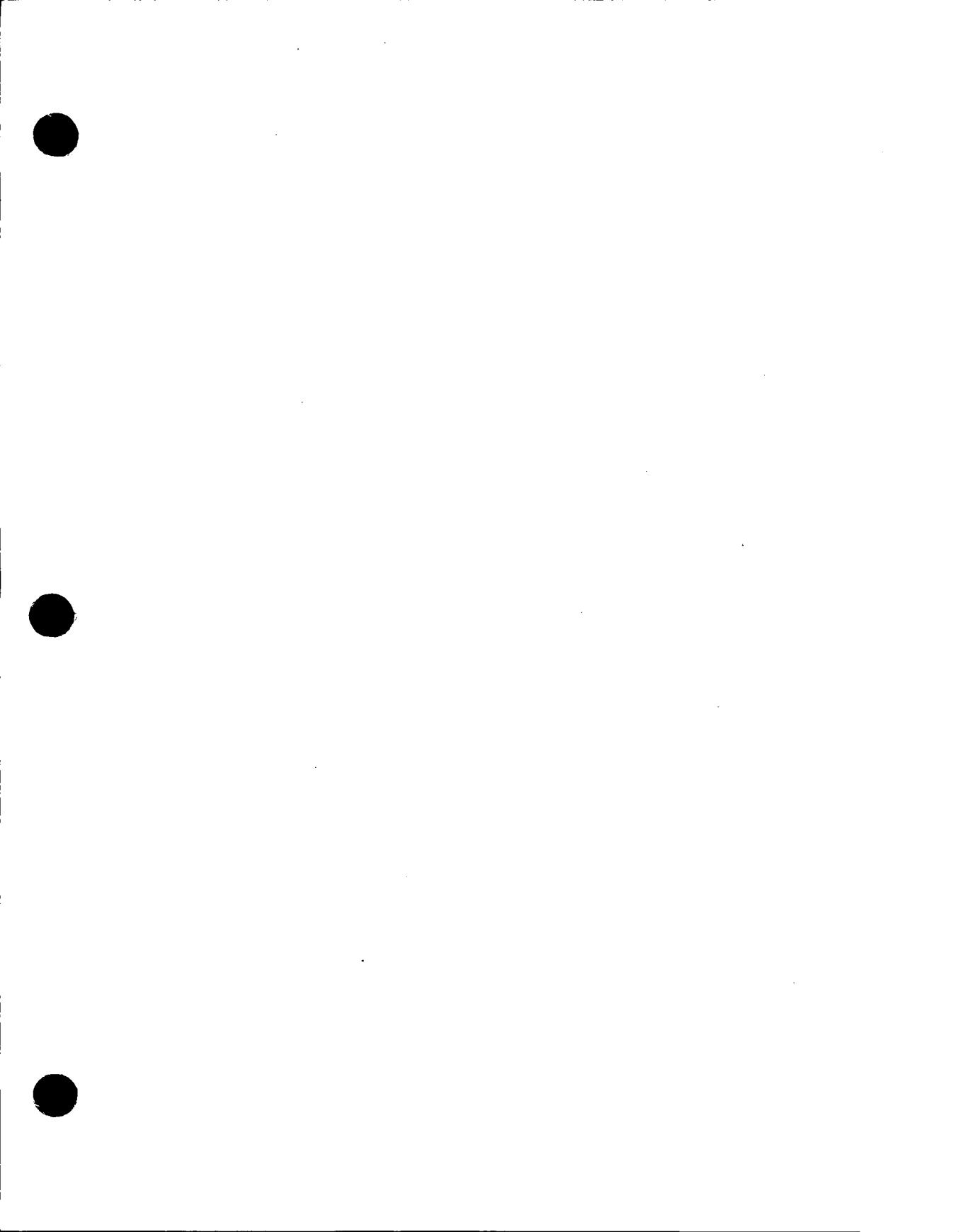
## Documentation Errata

### -H <asmb\_symfile> assembler option

This option is correctly shown in the assembler command line syntax of the UNIX/DOS User's Guide. The description of this option has been inadvertently omitted. The -H option has the same behavior as the -h option except it requires a file name for the HP-OMF asmb\_symbol output.

### -F <format> linker option

There is an error in the description of this option in the UNIX/DOS Users's Guide. The manual incorrectly states that S1 or S2 can be given as a legal format. For S-record output, only S, without a number can be specified.





***Microtec  
Research***

**2350 Mission College Blvd.**

**Santa Clara, CA 95054**

**Tel. 408.980.1300**

**Toll Free 800.950.5554**

**FAX 408.982.8266**



*Microtec  
Research*

***DOS Memory  
Management  
Supplement***

101067-001

## TRADEMARKS

Microtec® and Paragon® are registered trademarks of Microtec Research, Inc.

MRI ASM™, MRI C™, MRI FORTRAN™, MRI Pascal™, MRI XRAY™, Source Explorer™, XHM302™, XRAY™, XRAY Debugger™, XRAY In-Circuit Debugger™, XRAY In-Circuit Debugger Monitor™, XRAY MasterWorks™, XRAY/MTD™, XRAY180™, XRAY29K™, XRAY51™, XRAY68K™, XRAY80™, XRAY86™, XRAY88K™, XRAY960™, XRAYG32™, XRAYH83™, XRAYH85™, XRAYM77™, XRAYSP™, XRAYTX™, and XRAYZ80™ are trademarks of Microtec Research, Inc.

Other product names mentioned in this document are trademarks or registered trademarks of their respective companies.

## RESTRICTED RIGHTS LEGEND

If this product is acquired under the terms of a: *DoD contract*: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of 252.227-7013. *Civilian agency contract*: Use, reproduction, or disclosure is subject to 52.227-19 (a) through (d) and restrictions set forth in the accompanying end user agreement. Unpublished rights reserved under the copyright laws of the United States.

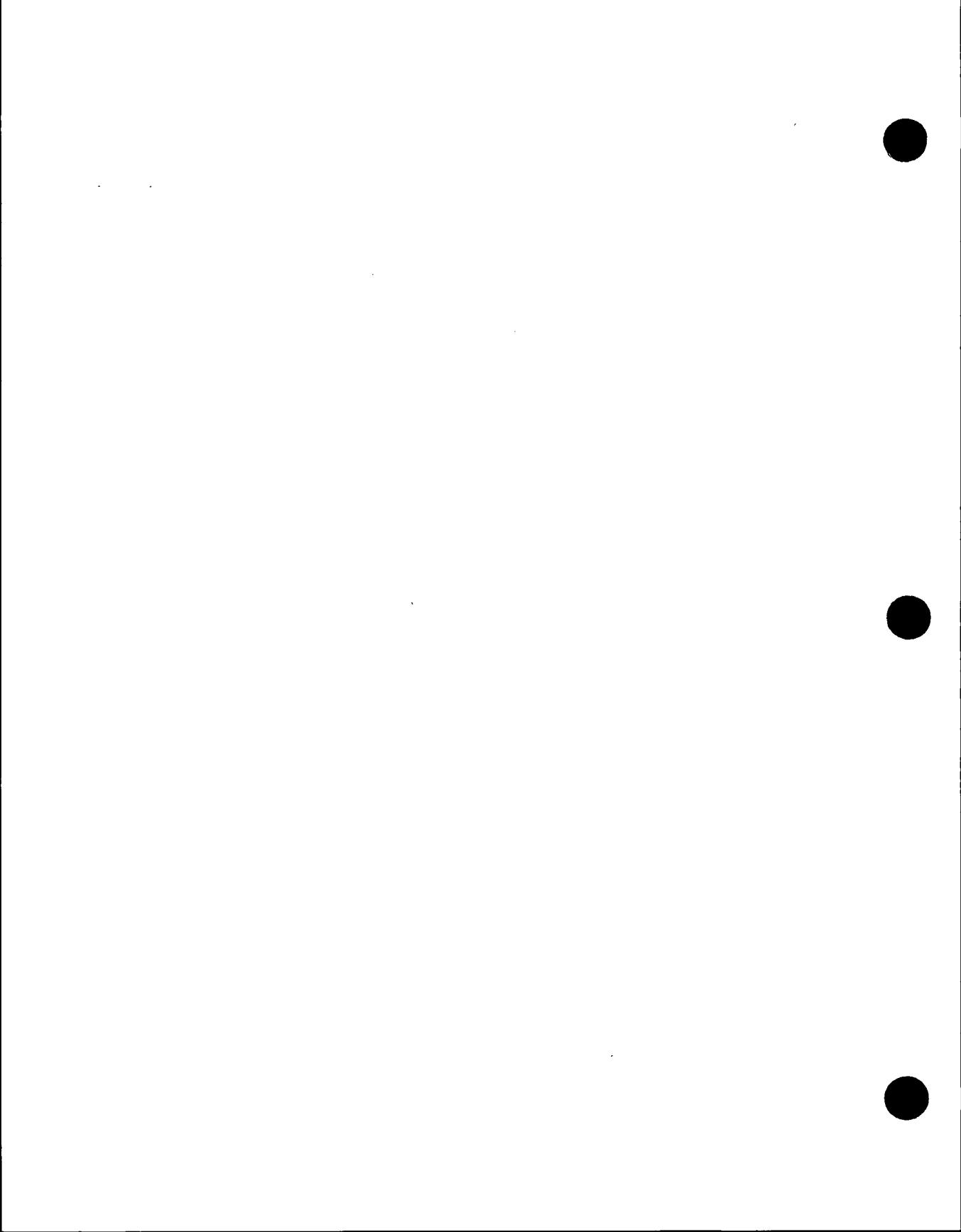
MICROTEC RESEARCH, INCORPORATED  
2350 MISSION COLLEGE BLVD.  
SANTA CLARA, CA 95054

Copyright © 1993 Microtec Research, Inc. All rights reserved. No part of this publication may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical or otherwise, without prior written permission of Microtec Research, Inc.

Revision History

---

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.	01/93	L.L.



# Preface

---

## About This Supplement

The *DOS Memory Management Supplement* consists of:

Chapter 1, *Introduction*, describes how the Rational Memory Management System, a memory manager used for all Microtec Research products, affects your product.

Appendix A, *Error Messages*, lists the error messages produced by the Rational Memory Management System.

## Questions and Suggestions

To help us respond to questions or suggestions regarding this product, we have provided two response forms in the back of this documentation.

The first form is a Reader's Response sheet, which is used to help us correct and improve our documentation. We always appreciate your comments, and by completing this form, you can participate directly in the revisions of our publications. The Reader's Response sheet is directed to Technical Publications.

The second form is a Software Performance Report, which should be completed if you encounter any errors or problems with Microtec Research software. This report is directed to Technical Support.



# Contents

---

## Preface

About This Supplement .....	v
Questions and Suggestions .....	v

## 1 Introduction

DOS Memory Management System .....	1-1
Setting Memory Parameters for XRAY .....	1-1
Changing the Memory Configuration .....	1-1

## Appendix A: Error Messages

Overview .....	A-1
Error Messages .....	A-2



# Introduction 1

---

## DOS Memory Management System

Microtec Research products use the Rational Memory Management System on PC hosts. Some PC host systems require that the **DOS16M** environment variable be set to indicate the location of extended memory on your host if the default setting is incorrect.

For NEC PC-9801 series machines, the **DOS16M** environment variable must be set as shown below:

```
set DOS16M=1 @start_address-end_address
```

The *start\_address* and *end\_address* are decimal or hexadecimal numbers. Hexadecimal format requires a **0x** prefix. The number may end in a **K** or **k** to indicate an address in kilobytes or an **M** or **m** to indicate megabytes. If no suffix is specified, the default is kilobytes. An example of setting the **DOS16M** environment variable on a NEC PC-9801 is shown below:

```
set DOS16M=1 @1m-2m
```

The Rational diagnostic tool **PMINFO** displays performance statistics for real- and protected-mode memory. If you encounter memory errors, such as insufficient memory, this utility can help our Technical Support Department identify and solve problems related to your configuration.

## Setting Memory Parameters for XRAY

The **XHSproduct.VMC** file (where *product* is the name of your product) can be used to modify your memory configuration. Use any ASCII text editor to create this file and enter the appropriate parameters. The various parameters are described in the next section.

XRAY searches for the **XHSproduct.VMC** file using the path(s) specified in the **XRAYLIB** environment variable.

## Changing the Memory Configuration

Setting the following parameters in your **XHSproduct.VMC** file will alter your memory configuration:

**SWAPSIZE=***size*      Specifies the size of the swap file in kilobytes. Specifying a large initial swap file size may speed up load times of large programs.

**VIRTUALSIZE=***n*      Specifies the maximum size of the VMM-managed space in kilobytes. The maximum and default value for *n* is 4096. Reducing this value may impact performance and/or could prevent XRAY from executing.

Setting the **SWAPSIZE** parameter will override the default **VIRTUALSIZE** parameter. Only one of these two parameters may be used at any one time.

**Example:**

**SWAPSIZE=2048**

# Error Messages: Appendix A

---

## Overview

This appendix lists the messages produced by the Rational Memory Management System on PC host systems. Messages are tagged with a prefix indicating where the error occurred and sometimes an error number. All prefixes are removed from the messages listed in this appendix.

Messages are listed alphabetically based on the first significant word (words like the, an, a, etc. are not considered significant).

### Note

If you encounter an error message not listed in the documentation for your product or in this appendix, call Microtec Research Technical Support. Be prepared to identify the error message, any error number that you received, and the circumstances which caused the error.

## Error Messages

### **8042 timeout**

This message indicates a keyboard problem. On most IBM AT-compatible computers, DOS/16M must use the 8042 keyboard processor to enable access to extended memory. This message means that a command was sent to the 8042 to enable or disable access, but the 8042 has not acknowledged the command. This message almost always indicates that DOS/16M is trying to use an AT switch mode on a non-AT-compatible computer. Use the **DOS16M** DOS environment variable to specify a correct switch mode.

### **cannot free memory**

DOS/16M is terminating and trying to return its memory to DOS, but DOS has signaled an error. Probably, the DOS memory allocation structures have been corrupted. Reboot your computer.

### **cannot initialize VCPI**

DOS/16M has detected that VCPI is present, but VCPI returns an error when DOS/16M tries to initialize the interface. Your machine configuration is incorrect. Run **pminfo** to check available memory for VCPI.

### **cannot run under OS/2**

DOS/16M programs can actually run under OS/2, but since OS/2 does not let DOS programs access extended memory, they can only use real memory. There is not enough memory for the DOS/16M program to run.

### **can't allocate internal tables**

There is not enough physical memory for the swap file allocation table. Run **pminfo** to check memory availability. To free more memory, remove memory-resident programs (TSRs) and any unnecessary system software. You can also decrease the swap file size or install additional physical memory.

**can't allocate tables**

There is not enough physical memory to start VMM. Run **pminfo** to check memory availability. To free more memory, remove memory-resident programs (TSRs) and any unnecessary system software. You can also install additional physical memory.

**can't create swapfile: *filename***

The environment variable **MRI\_product\_TMP** (where *product* is the name of your product) or **TMP** may be pointing to a directory that does not exist. Check your compiler or assembler manual for more information.

**Can't find loader for file [code] — *filename***

This message indicates that the program was unable to find a loader that can load an executable of this format. The paths searched for the loader are:

1. Paths specified by the **XRAYLIB** environment variable
2. Paths specified by the **PATH** environment variable

**Can't locate DOS extender**

You may get this error if you change the name of the loader to something else. The paths searched for the loader are:

1. Paths specified by the **XRAYLIB** environment variable
2. Paths specified by the **PATH** environment variable

**can't open configuration: *filename***

VM finds a **.VMC** file, but cannot open it. The message is a result of a DOS error. The file may be corrupted. Examine your **.VMC** file for problems. Re-create the file if necessary.

**can't open file *filename***

DOS may not have enough file units configured (increase the **FILES=** entry in **CONFIG.SYS**).

**computer must be AT- or PS/2- compatible**

The computer has a 286 CPU, but does not have BIOS or hardware support for protected-mode operation.

**DPMI host error (cannot lock stack)**

This error indicates insufficient memory under DPMI. Run **pminfo** and check memory available for DPMI.

Under Windows, you might try making more physical memory available by eliminating or reducing any RAM drives or disk caches. You might also try editing **DEFAULT.PIF** so that at least 4MB of XMS memory is available to non-Windows programs.

**DPMI host error (need 64K XMS)**

This error indicates insufficient memory under DPMI. Run **pminfo** and check memory available for DPMI.

Under Windows, you might try making more physical memory available by changing parameters in your **CONFIG.SYS** file or reducing any RAM drives or disk caches. You might also try editing **DEFAULT.PIF** so that at least 4MB of XMS memory is available to non-Windows programs.

**DPMI host error (possibly insufficient memory)**

This error indicates insufficient memory under DPMI. Run **pminfo** and check memory available for DPMI.

Under Windows, you might try making more physical memory available by changing parameters in your **CONFIG.SYS** file or reducing any RAM drives or disk caches. You might also try editing **DEFAULT.PIF** so that at least 4MB of XMS memory is available to non-Windows programs.

**insufficient DOS memory**

There is not enough low memory for your program. Run **pminfo** to check physical memory availability. Remove memory-resident programs (TSRs) and any unnecessary system software.

**insufficient physical memory**

Run pminfo to check physical memory availability. To free more memory, remove memory-resident programs (TSRs) and any unnecessary system software. You can also install additional physical memory.

**insufficient virtual memory**

You have run out of virtual memory. Increase your virtual space by altering the value of the VIRTUALSIZE environment variable.

**no available memory**

The virtual size is too small or there is not enough physical memory. Run pminfo to check physical memory availability. To free more memory, remove memory-resident programs (TSRs) and any unnecessary system software. You can also install additional physical memory. To increase the virtual size, increase the value of the VIRTUALSIZE environment variable.

**no DOS memory for transparent segment**

You are attempting to allocate a transparent segment, and DOS has indicated that there is not enough memory available for the segment. Remove memory-resident programs (TSRs) and any unnecessary system software or free some DOS memory.

**no memory for VCPI page table**

When DOS/16M sets up for VCPI operation, it needs additional memory for the 386 hardware page tables. There is not enough DOS memory available for the tables.

**not a DOS/16M executable *filename***

The .EXP file or .EXE file you are trying to load is not a DOS/16M file or has been corrupted in some way. Reinstall the original file.

**not enough disk space**

There is not enough disk space for the swap file to expand. The program exits and all data is lost. Increase the space on your hard disk by removing unnecessary files.

**not enough disk space for swapping - required bytes: *bytes***

DOS error. There isn't enough disk space to create the swap file (*bytes* indicates the total number of bytes you need in the swap file). Delete files or decrease the **VIRTUALSIZE** or **SWAPSIZE** variables.

**not enough extended memory**

Certain linkers require that the entire program be loaded in one physical memory block. This message indicates that there is not enough extended memory to load the program. You need more extended memory.

**not enough memory to load program**

The program is larger than available memory. Either provide more memory or remove memory-resident programs.

**not enough swap space**

The swap file is not big enough. If you have set the **SWAPSIZE** parameter in the **.VMC** file, increase its value. If you are using **VIRTUALSIZE**, increase its value.

**out of memory**

VMM is not reserving enough virtual memory for your program. Increase the **VIRTUALSIZE** parameter in your **.VMC** file or in your environment.

**out of swap space**

The swap file is not big enough. If you have set the **SWAPSIZE** parameter in the **.VMC** file, increase its value. If you are using **VIRTUALSIZE**, increase its value.

**premature EOF**

When loading your protected mode program into memory, the loader found the end of the file before it was expected. Most likely, the file has been corrupted. Reinstall the original file.

**program larger than virtual memory**

VM is not reserving enough virtual memory for your program. Increase the **VIRTUALSIZE** parameter in your .VMC file.

**protected mode available only with 286, 386, or 486 CPU**

DOS/16M requires an 80286, 80386, or 80486 CPU. It cannot run on an 8086, 8088, or 80186 CPU.

**requires DOS 3.0 or later**

Upgrade your DOS version.

**swap out**

The swap file is not big enough. If you have set the **SWAPSIZE** parameter in the .VMC file, increase its value. If you are using **VIRTUALSIZE**, increase its value.

**Syntax — DOS4GW *executable.xxx***

This message occurs when you execute the DOS4GW program without giving the name of an executable program on the command line.

**system software does not follow VCPI or DPMI specifications**

Some memory-resident program has put your 386 or 486 CPU into Virtual 8086 mode in order to provide special memory services to DOS programs, such as EMS simulation (EMS interface without EMS hardware) or high memory. In this mode, it is not possible to switch into protected mode unless the resident software follows a standard that DOS/16M supports. VCPI (Virtual Control Program Interface) is one such standard and is followed by QEMM from Quarterdeck, 386<sup>MAX</sup> from Qualitas, and CEMM from Compaq, among others. If you have an incompatible memory-resident program (such as EMM386), turn off this program in your CONFIG.SYS file and reboot your computer.

**The DOS16M.386 virtual device driver was never loaded**

You started a DOS/16M memory-resident program (TSR) before you started Windows 3.0 in 386 enhanced mode, but the TSR could not find the device driver (**DOS16M.386**) needed to support it. You can run Windows in real or standard

mode, you can unload the TSR before starting Windows, or you can make sure the TSR can find **DOS16M.386**.

**unsupported DOS16M switchmode choice *switchmode\_value***

The value specified for the **DOS16M** environment variable is one that DOS/16M cannot interpret. Use the **DOS16M** DOS environment variable to specify a correct switch mode.

**.VMC: specify SWAPSIZE or VIRTUALSIZE**

You have specified both **SWAPSIZE** and **VIRTUALSIZE** in the **.VMC** file. Remove the specification for one or the other.

**.VMC: SWAPFILE specification conflict**

You have specified conflicting swap file requirements. For example, **SWAPSIZE=1024** and **SWAPFILESIZE=600**.

**.VMC: unrecognized option: *name***

You have used an unrecognized option in your **.VMC** file. Check the spelling in the file.

**You must specify an extended memory range (SET DOS16M=)**

This computer is not IBM AT-compatible, and has no protocol for managing extended memory. Add an extended memory range specification to the **DOS16M** environment variable. For example the command, **SET DOS16M=1 @1M - 3M**, sets the machine type to 1 (NEC 9801) and specifies that DOS/16M will only use memory from 1MB to 3MB.

# Index

---

## Symbols

- .VMC file 1-1
- SWAPSIZE parameter 1-2
- VIRTUALSIZE parameter 1-2

## D

- Diagnostics
- PMINFO program 1-1
- DOS16M environment variable 1-1

## E

- Environment variables
- DOS16M 1-1
- XRAYLIB 1-1
- Error messages A-1
- Extended memory
- DOS16M environment variable 1-1

## M

- Memory
- configuration
- .VMC file 1-1
- extended 1-1
- Memory manager
- Rational 1-1
- Messages, error A-1

## N

- NEC PC-9801 setup 1-1

## P

- PMINFO program 1-1

## R

- Rational memory manager 1-1
- Reader's response sheet v
- Response forms v

## S

- Software performance report v
- SWAPSIZE parameter
- .VMC file 1-2

## V

- VIRTUALSIZE parameter
- .VMC file 1-2
- .VMC file 1-1

## W

- Warning messages A-1

## X

- XHS(product).VMC file 1-1
- XRAYLIB environment variable 1-1



## Reader's Response

Please complete and return the following Reader's Response form to help us improve our documentation. Your comments and suggestions are always appreciated.

Product Name \_\_\_\_\_

Manual Date/Version \_\_\_\_\_

Your Name \_\_\_\_\_

Title \_\_\_\_\_

Company \_\_\_\_\_

Address \_\_\_\_\_

Phone Number \_\_\_\_\_

What is your level of programming experience?

Advanced

Advanced

Intermediate

Intermediate

Beginner

Beginner

Circle the number which best expresses your rating of the information in this manual (*5 is the highest rating*):

Complete?    1    2    3    4    5

Easy to understand?    1    2    3    4    5

1    2    3    4    5

Accurate?    1    2    3    4    5

Helpful examples?    1    2    3    4    5

1    2    3    4    5

Well organized?    1    2    3    4    5

Helpful procedures?    1    2    3    4    5

1    2    3    4    5

Easy to Find?    1    2    3    4    5

At an appropriate level?    1    2    3    4    5

1    2    3    4    5

Easy to Read?    1    2    3    4    5

Overall rating?    1    2    3    4    5

1    2    3    4    5

3.) Which programming language do you use?

---

4.) Does the manual provide all the necessary information? If not, what is missing?

---

5.) Are more examples needed? If so, where?

---

6.) Did you have any difficulty understanding descriptions or wording? If so, where?

---

7.) Which sections of this manual are the most important? The most helpful?

---

8.) What are the best features of this manual?

---

9.) What are the areas in this manual that need improvement?

---

10.) Did you find any errors in this manual? (Specify section and page number.)

---



## **Software Performance Report Instructions**

If you encounter problems or errors when using Microtec Research software products (including documentation), you can send a Software Performance Report (SPR):

- Electronically through e-mail to **support@mri.com**

or

- By completing the SPR form on the next page and returning it to:

**Microtec Research  
Software Performance Reports  
2350 Mission College Boulevard  
Santa Clara, CA 95054**

All SPRs are directed to our Technical Support Department for analysis and response.

To make it easier for us to respond quickly to your problem, please include the following:

1. Give a complete description of the problem, error, or suggestion, including the exact wording of any error messages.
2. If possible, isolate the problem by providing a small example.
3. If you are mailing or FAXing an SPR and your error example is longer than one page of source code, please provide all information on a tape or floppy disk.
4. Include command files, listings, load maps, include files, data files, and all other relevant material with the message.
5. If applicable, send sample input and output listings.
6. If the output is from a compiler, send a mixed (source/assembly code) listing.
7. Be sure to include the following information:
  - Product name and version number
  - Serial number
  - Name of contact
  - Company purchased from
  - Company name and a shipping address
  - Phone number and/or FAX number
  - Host machine
  - Host operating system and version number
  - Customer impact

All communications will be acknowledged the next working day.



## Software Performance Report

Product:	Customer:
Version:	Company Name/Address:
Serial Number:	
Purchased From:	
	Phone Number:
	E-Mail Address:

Host:       Apollo       PC  
               DECstation       Sun-3  
               HP 9000 Series 300/400       Sun-4/SPARC  
               HP 9000 Series 700       VAX/VMS  
               IBM RS/6000       Other \_\_\_\_\_

Operating System \_\_\_\_\_ Version \_\_\_\_\_

Report Type: \_\_\_\_\_ Customer Impact: \_\_\_\_\_

(Attach additional pages if necessary to complete the items below.)

Problem Description:

Commands or lines of code used to duplicate problem:

Workaround:





Responsible  
for

Local Production Control Project

Supervising Engineer

Tele 4040-3333-1380

Local Office 404-333-3334

Local Office 404-333-3335



***Microtec  
Research Inc.***

2350 Mission College Blvd.

Santa Clara, CA 95054

Tel. 408.980.1300

Toll Free 800.950.5554

FAX 408.982.8266