Assembler/Linker/ Librarian

User's Guide and Reference for the 68000 Family

100113-013

Microtec
880 Ridder Park Dr.
San Jose, CA 95131

00

REV.	REVISION HISTORY	DATE	APPD.
-002	Updated from V 6.3 to V 6.4 the VAX/VMS and UNIX chapters and the Reference Manual.	6/88	L.C.
-003	Updated from V 6.4 to V 6.5 the VAX/VMS and UNIX chapters and the Reference Manual.	8/89	L.C.
	Added the IBM-PC/MS-DOS chapter.		
-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.
	Printed V 6.4 at 80% for new packaging.		
-006	Updated from V 6.5 to V 6.6 the VAX/VMS, UNIX, and IBM-PC/MS-DOS chapters and the Reference Manual.	2/90	L.C.
	Added the **LLEN** and **P=68332** assembler command line flags.		
-007	Updated to Version 6.8A.	9/91	L.L.
-008	Updated to Version 6.9.	8/92	D.C.
-009	Updated to Version 7.0 and removed the VAX/VMS chapter.	12/93	E.M.
-010	Updated for 68K release 5.C	1/95	D.G.
-012	DOS Installation section removed and User's Guide and Reference manuals combined.	12/96	M.G.
-013	Updated to Version 7.1.	8/97	E.C./J.Y.

Contents

4 Instructions and Address Modes

Figures

Tables

Index

Preface

About This Manual

This manual serves as both a user's guide and reference for the Microtec assembler for the 68000 family of microprocessors. The *User's Guide* section provides host computer and operating system-specific descriptions of how to invoke the assembler, linker, and object module librarian. The *Reference* section describes the assembler syntax and directives, linker commands, and librarian commands and gives sample sessions, output listings, and error and warning messages for each one.

The *User's Guide* section of this manual is organized as follows:

- 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 chapter lists and describes the operating system-specific command syntax and command line options for each tool.

The *Reference* section of this manual is organized as follows:

- Chapters 3 and 4 describe the assembly language for the 68000 family of microprocessors, including assembly language programming syntax, expressions, instructions, and their operands.
- Chapters 5 through 9 describe the ASM68K Assembler, including the special assembler controls and all of the directives and codemacros that it supports. Chapter 8 shows an assembler session with sample input command files and output listings.
- Chapters 10 through 12 describe LNK68K Linker concepts, commands, and a sample linker session with input command files and output listings.
- Chapters 13 through 15 describe LIB68K Object Module Librarian concepts, commands, and example sessions.

The *Appendices* section of this manual is organized as follows:

- Appendix A, *ASCII Character Set,* contains the name, octal, decimal, and hexadecimal values of ASCII 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, *IEE2AOUT Error Messages*, describes the warning and error messages produced by IEE2AOUT.
- Appendix F, *Glossary*, contains a glossary of some assembler terms.
- Appendix G, *Object Module Formats*, describes output module formats generated by the linker.
- Appendix H, *C++ Support*, describes how the assembler, linker, and librarian were modified to support C++.
- Appendix I, *HP 64000 Development System Support*, describes how the ASM68K assembler package interacts with the HP 64000 Development System.

This documentation assumes a working knowledge of the Motorola 68000 family of microprocessors. For background information, see the list of references in the section *Related Publications* in this preface.

Microprocessor References

The 68000 family of microprocessors includes the 68000, 68EC000, 68HC000, 68HC001, 68008, 68010, 68020, 68EC020, 68030, 68EC030, 68040, 68060, 68EC040, 68EC060, 68302, 68851, 68881, 68882, and the CPU32/CPU32+ family microprocessors: 68330, 68331, 68332, 68333, 68340, 68349, 68360. This document refers to these microprocessors as 68000 family microprocessors.

Related Publications

This documentation 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 the following Motorola publications:

- *Programmer's Reference Manual*, MC68000PM/AD.
- *68000 Microprocessor User's Manual*, MC68000UM/AD.

- *68020 Microprocessor User's Manual*, MC68020UM/AD.
- *MC68881/MC68882 Floating-Point Coprocessor User's Manual*, MC68881UM/AD.
- *68030 Enhanced 32-Bit Microprocessor User's Manual*, MC68030UM/AD.
- *MC68040 Enhanced 32-Bit Third Generation Microprocessor User's Manual*, MC68040UM/AD.
- *MC68060 Enhanced 32-Bit Third Generation Microprocessor User's Manual*, MC68060UM/AD.
- *68851 Paged Memory Management Unit User's Manual*, MC68851UM/AD.
- *CPU Central Processor Unit Reference Manual*, CPU32RM/AD.
- *M68000 Family Reference*, MC68000FR/AD.

For further information about the Microtec extended IEEE-695 format, refer to:

- *IEEE-695 Object Module Format Specification*, Microtec, December 1992.

This documentation 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.
- *AT&T C++ Library Manual Pages*, AT&T Bell Laboratories, 1990.
- *C++ for C Programmers*, by Ira Pohl, 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 Dewheurst and Kathy Stark, Prentice-Hall, 1989.

For information about other Microtec 68000 family toolkit components, refer to the following Microtec publications:

- *C Compiler User's Guide and Reference for the 68000 Family*

  This document describes how to use the MCC68K ANSI C Cross Compiler.

- *C++ Compiler User's Guide and Reference for the 68000 Family*

  This document describes how to use the CCC68K C++ Compiler.

- *XRAY Debugger Documentation Set*

  This documentation set describes how to use the XRAY68K Debugger.

- *XRAY In-Circuit Debugger Monitor Documentation Set*

  This documentation set describes how to use the XDM68K XRAY In-Circuit Debugger Monitor with the XRAY Debugger.

- *HP 64700 ICE Supplement for XRAY Documentation*

  This document describes how to set up and use the version of the XRAY Debugger that works in conjunction with the Hewlett-Packard HP 64700 in-circuit emulator for the 68000 family of microprocessors.

- *Product Installation (UNIX Hosts)*

  This document describes how to install and manage licenses for Microtec products that use Flexible Licensing.

Notational Conventions

This manual uses the notational conventions shown in Table P-1 (unless otherwise noted).

Table P-1. Notational Conventions

Symbol	Name	Usage
{ }	Curly Braces	Enclose a list from which you must choose an item.
[ ]	Square Brackets	Enclose optional items.
. . .	Ellipsis	Indicates that you may repeat the preceding item zero or more times.
|	Vertical Bar	Separates alternative items in a list.
	Punctuation	Punctuation such as commas (**,**) and colons (**:**) must be entered as shown.
	`Typewriter Font`	Represents code or user input in interactive examples.
	Italics	Represents a descriptive item that should be replaced with an actual item.
	Bold	Represents elements that need to stand out from the main body of text.

Questions and Suggestions

Microtec is committed to providing its customers with quality software development and RTOS tools and support services. Our commitment continues beyond your purchase of the product throughout your development life cycle.

If you have questions or suggestions regarding this product, please contact your Microtec support representative. Contact numbers are listed on the back cover of this document.

Introduction 1

Components of the Microtec ASM68K Assembler Package

The Microtec 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 family of microprocessors. The following sections describe the components of the Microtec ASM68K assembler package.

Assembler

The Microtec ASM68K Assembler converts assembly language programs into 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 68000 family assemblers. Many additional assembler 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 binds the object modules together into a single object module, which is then suitable for downloading and execution on any 68000 family microprocessor.

Linker

The Microtec LNK68K Linker combines relocatable object modules into a single absolute object module. You can generate object modules in the following formats (some formats may require the use of one of the supplied conversion programs):

- Microtec extended IEEE-695 format
- Motorola S-record format
- Hewlett-Packard 64000 format (HP-OMF)
- Sun Microsystems **a.out** format

The linker also supports the combining of multiple relocatable object modules into a single relocatable module, which you can subsequently relink 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 locations 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 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 the linker will subsequently use. You can add, delete, replace, and extract modules as well as obtain a listing of library contents.

ASM68K Features

Significant features of ASM68K include:

- Support of the 68000 family Central Processing Unit (CPU) instructions
- Support of the 68000 family Floating-Point Unit (FPU) instructions
- Support of the 68000 family Memory-Management Unit (MMU) instructions
- Versatile directive set
- Address constants specified easily
- Assembly-time relative addressing
- Several different data creation statements
- Storage reservation statements
- Character codes specified in ASCII or EBCDIC
- Flexible assembly listing control statements
- Comments and remarks used for documentation

- Symbolic and relative address assignments and references
- Symbolic or cross-reference table listing generation
- Production of object modules in IEEE-695 format, Motorola S-record format, or HP-OMF format
- Absolute or relocatable (incremental linking) object modules generation
- Support of forward references
- Conditional assembly facility
- User-defined macro facility
- High-level debugging information for the Microtec XRAY68K Debugger
- Support for case-sensitive symbols
- Manufacturer-compatible symbolic machine operation codes (opcodes, directives)
- Complex relocatable expression evaluation
- C and Pascal-like run-time structured control directives
- Nested line numbers in listing
- Flexible assembly listing control statements
- C++ support
- **cpp**-like directives for conditional assembly and expression definitions

These features will help you produce well-documented, modular, working programs in a minimum of time.

Other Microtec Products

The following are other products produced by Microtec for embedded systems development.

MCC68K C Compiler

The Microtec C compiler converts both ANSI C and traditional (K&R) C source programs into tight, efficient assembly language code for 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.

CCC68K C++ Compiler

The Microtec C++ compiler converts C++ source code into tight, efficient assembly language code for the ASM68K Assembler. It accepts several different dialects of C++, including the subsets accepted by **cfront** versions 2.0 and 3.0, and supports features such as templates and exception handling. You can use the C++ compiler to create ROMable programs and generate debugging information for the XRAY Debugger.

XRAY Debugger

The Microtec XRAY68K 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 values 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.

The XRAY68K Debugger'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 windows. 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 fix your source code.

Data Flow

Figure 1-1 shows the components of the Microtec ASM68K Assembler package and the ways in which they communicate with each other and with the host computer system.

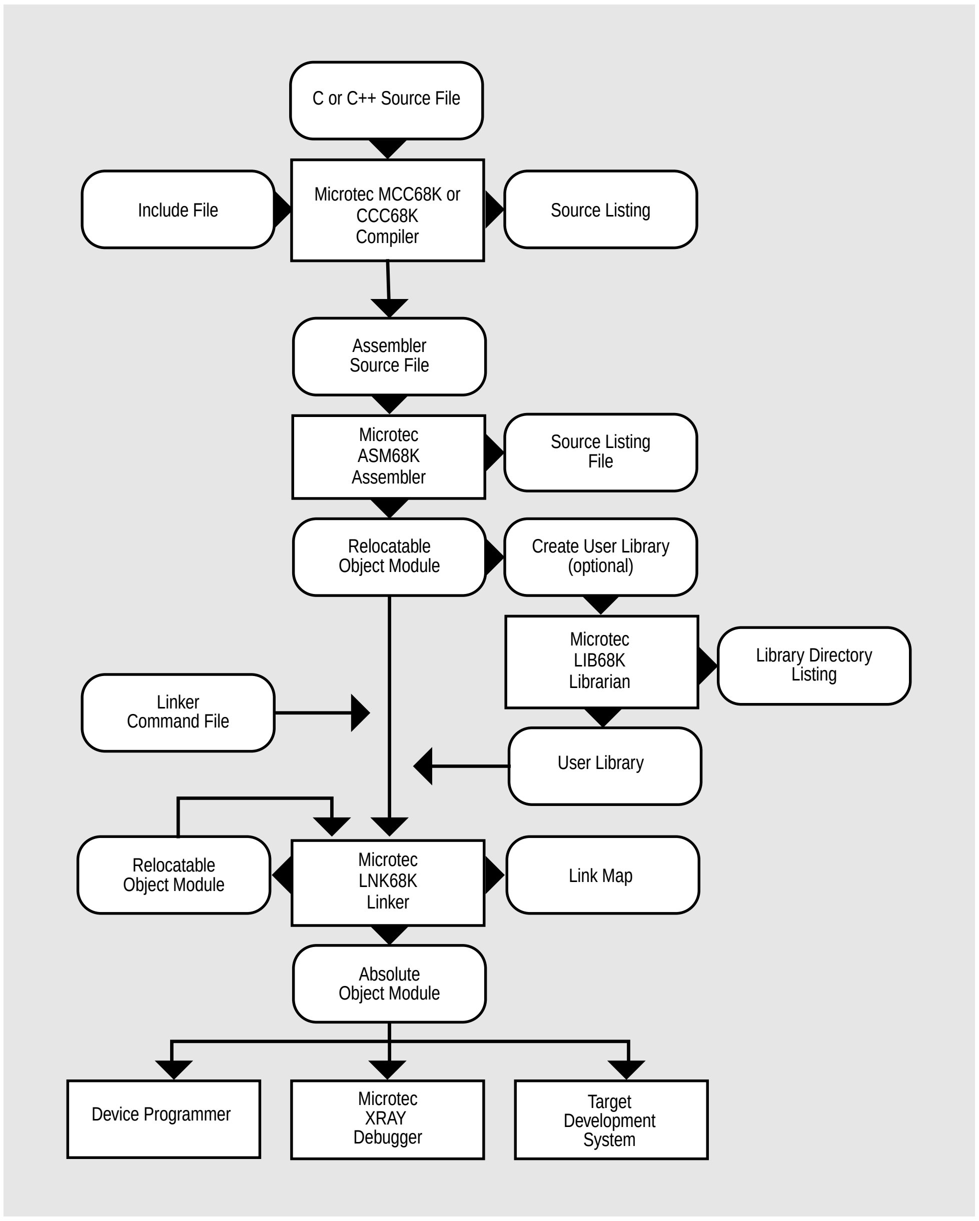

C or C++ Source File
Include File
Microtec MCC68K or CCC68K Compiler
Source Listing
Assembler Source File
Microtec ASM68K Assembler
Source Listing File
Relocatable Object Module
Create User Library (optional)
Microtec LIB68K Librarian
Library Directory Listing
Linker Command File
User Library
Relocatable Object Module
Microtec LNK68K Linker
Link Map
Absolute Object Module
Device Programmer
Microtec XRAY Debugger
Target Development System

Figure 1-1. Components of the Microtec ASM68K Assembler Package

UNIX/DOS User's Guide 2

Introduction

This chapter describes the basic use of the assembler, linker, and object module librarian on all hosts running UNIX or DOS operating systems. The chapter is divided into a section for each tool; each section contains descriptions of the following information, where appropriate:

- Invocation syntax and command line options
- Command line flag descriptions
- Input and output file name defaults
- Environment variables
- Invocation examples

The last section of the chapter describes the return codes generated by the tools and conversion programs that come with the assembler package.

Examples in this chapter are identified by operating system name. The name UNIX refers to any UNIX or UNIX-like operating system, such as SunOS or HP-UX. The name DOS refers to both MS-DOS and PC-DOS systems.

ASM68K Assembler

The Microtec ASM68K Assembler produces object code typically for later use by the LNK68K Linker and LIB68K Librarian.

Macro Preprocessor

The ASM68K Assembler has a macro preprocessor that is similar to the standard C preprocessor (cpp). It enables assembly language programmers to make use of the following C-style features:

- File inclusion
- Conditional assembly
- Defining of symbols as constant expressions
- Undefining of symbols

The macro preprocessor directives are described in the section *Macro Directives* in Chapter 7, *Macros*.

Assembly Language 3

Introduction

A 68000 family microprocessor executable program typically (but not always) consists of a sequence of 32-bit binary numbers contained in memory. These numbers represent 68000 family 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:

- Instruction statements
- Directive statements
- Macro statements
- Comment statements

The syntax for these statements is shown below:

Syntax:

[*label*] [*operation* [*operand*[,*operand*]...]] [*comment*]

Description:

label	The label field assigns a memory address or constant value to the symbolic name contained in the field. The label field can 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.

It is legal for a label to be the only field in a statement. The first 512 characters of a label are significant.

Labels are case-sensitive by default. Case sensitivity can be turned off 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.

Note

The *operand* field cannot have any whitespace within it; everything after a space in an *operand* field is treated as a comment.

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 (**;**). A comment can be on a line by itself. It can begin in any column and terminates at the end of the line.

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 or opcode) 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 (*label*). 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 `MOVE`.
`MOVE`	Is a symbolic opcode (*operation*) representing the bit pattern of the load instruction.
`MEM`	Is a symbol (*operand*) representing a memory address.
`D2`	Is a reserved symbol (part of *operand*) representing a data register.

Directive Statement

A directive statement is a control statement to the assembler. It is not translated into a machine instruction.

Example:

```
ABAT  DC     DELT
```

where:

`ABAT`	Is a symbol (*label*). 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 `DC` directive.
`DC`	Is a directive (*operation*) that instructs the assembler program to allocate two bytes of memory.
`DELT`	Is a symbol (*operand*) representing an address. The address will be placed into the two bytes allocated by the `DC` directive.

Macro Statement

A macro statement is a call to a sequence of instructions. The call can be made from any part of the program. The requirements for creating macros are described in Chapter 7, *Macros*, in this manual. 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 the initial macro declaration, 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 assembly language statements. A comment statement begins with a semicolon (**;**), asterisk (*****), or exclamation point (**!**). Comments can begin in any column and terminate at the end of the line.

Example:

```
; This is a comment statement.
```

Blank lines are also treated as comment statements.

Note

An asterisk (*****) is considered a comment statement if it is the first character on a line. If it is not the first character, the asterisk can be interpreted as the assembly program counter or multiplication operator.

Symbolic Addressing

When you write statements in assembly language, the assembler expresses the machine operation code symbolically.

For example, the machine instruction that rotates the contents of a register 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 family, the assembler defines 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. For example, if the instruction in the previous example is stored at the address 128, then the symbol `SAVR` will be set to the value 128 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 branching to the instruction pointed to by the label, you could write:

```
JMP SAVR
```

When the branch instruction is translated into machine code by the assembler, the address of the destination instruction is placed in the address field of the branch 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 can be accomplished through the use of the plus (+) and minus (-) operators.

Example:

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

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 since the variation in the number of bytes per instruction can cause references to the wrong location. In the previous example, the `MOVE` instruction requires 2 bytes, and the `CMPI` instruction requires 4 bytes.

The assembler expansion of the previous 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, and grammar rules 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 3-1. 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. Alphabetical characters are case-sensitive by default. If case sensitivity is turned off, the case is not maintained.

Table 3-1. Special Characters Recognized by the Assembler

&	ampersand	-	minus sign
*	asterisk	%	percent sign
@	at sign	.	period
`	back quote (accent grave)	+	plus sign
\	backslash	#	pound
	blank	?	question mark
:	colon	}	right brace
,	comma	]	right bracket
$	dollar	)	right parenthesis
"	double quote	;	semicolon
=	equal sign	'	single quote
!	exclamation	/	slash
>	greater than		tab
{	left brace	~	tilde
[	left bracket	_	underscore
(	left parenthesis	^	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 512 characters of a symbol are significant, although use of symbols longer than 127 characters is not recommended. The first character in a symbol must be alphabetic or one of these special characters:

?	Question mark
.	Period
_	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-sensitive by using the **OPT CASE** 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 code your own symbols beginning with two question marks, specify the **OPT G** directive to make these symbols available for debugging.

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 3-2, are the symbolic register names used to denote the various hardware registers, expression operators, and the macro parameter count; they will not appear in a symbol table or in a cross-reference listing.

Table 3-2. Reserved Symbols

Register Category	Symbol
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
Cache symbols (68040/60 only)	IC, BC, DC
Control registers	CCR, PC, SR, USP
68010 registers	VBR, SFC or SFCR, DFC or DFCR

(cont.)

Table 3-2. Reserved Symbols (cont.)

Register Category	Symbol
68020 pseudo registers[a]	ZA0, ZA1, ZA2, ZA3, ZA4, ZA5, ZA6, ZA7, ZD0, ZD1, ZD2, ZD3, ZD4, ZD5, ZD6, ZD7, ZPC
68881/82 registers[a]	CONTROL, FP0, FP1, FP2, FP3, FP4, FP5, FP6, FP7, FPCR, FPSR, FPIAR, IADDR, STATUS
68851 registers[a]	AC, BAD0, BAD1, BAD2, BAD3, BAD4, BAD5, BAD6, BAD7, BAC0, BAC1, BAC2, BAC3, BAC4, BAC5, BAC6, BAC7, CAL, CRP, DRP, PCSR, SCC, SRP, TC, VAL
68EC030 registers[a]	CRP, AC0, AC1, ACUSR, SRP, TC
68030 registers[a]	CRP, MMUSR, SRP, TC, TT0, TT1
68040/60 registers[a]	CACR, DTT0, DTT1, ITT0, ITT1
68EC040/060 registers	CACR, DACR0, DACR1, IACR0, IACR1

a. Reserved only during this mode.

You can use the **EQU** directive to define keywords to represent these reserved symbols.

Example:

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

The previous 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 resides (i.e., code or data). 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.

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 integer, floating-point, character, or string 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. Table 3-3 lists the ranges per byte for constants.

Table 3-3. Constant Value Ranges

Number of Bytes	Value Range
1 (unsigned)	0 to 255
2 (unsigned)	0 to 65,535
3 (unsigned)	0 to 16,777,215
4 (unsigned)	0 to 4,294,967,295
1 (two's complement)	-128 to +127
2 (two's complement)	-32,768 to +32,767
3 (two's complement)	-8,388,608 to +8,388,607
4 (two's complement)	-2,147,483,648 to +2,147,483,647

Numbers 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** can be either +255 or -1 depending on the use. The assembler correctly recognizes 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 3-4 shows the available prefixes and their meanings. If no prefix is given, the default number base is used.

Table 3-4. Prefixes Used Before Constants

Suffix	Prefix	Constant
B	%	Binary
D		Decimal
Q, O	@	Octal
H	$	Hexadecimal

If the default base is hexadecimal, unprefixed values must begin with a digit from 0 to 9 to be recognized as a number (e.g., the hexadecimal number B19 would be considered a symbol or variable without a leading 0).

Examples:

```
t21_a   equu   %10101  ;binary
```

```
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     ;hexadecimal
t21_j   equ   15h     ;hexadecimal

        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     ;hexadecimal
        dc.b  15h     ;hexadecimal
```

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. The decimal floating-point number syntax is as follows:

Syntax:

`[+ | -]`*dec_value*`[.`*dec_value*`][[_]{E | e}[_][[+ | -]`*dec_value*`]]`

Description:

`+`	Specifies a positive floating-point number.
`-`	Specifies a negative floating-point number.
dec_value	A decimal value.
`e, E`	Specifies scientific notation. You can optionally place underscores (_) before or after the `E` to increase readability.

Examples:

`-1E-3`	(same as -0.001)
`3.14159`	(PI to 5 places)
`+3.333_E+5`	(same as 333300.0)
`0`	(zero)

Floating-point constants are permitted only in the **DC**, **DCB**, and **FEQU** directives. Figure 3-1 shows how floating-point constants are represented in hexadecimal and decimal formats. Additional information about floating-point constants appears in the description of the **DC** directive in Chapter 6, *Assembler Directives*, in this manual.

```
Microtec Research ASM68K   Version x.y     Wed Jul 22 13:27:07 1992    Page  1

Line Address
1                               SYM1  FEQU    :1234_5678_9ABC_DEF_
2                               SYM2  FEQU    :123456789ABCDEF
3
4    00000000 1234 5678 9ABC          DC.X    SYM1
              DEF0 0000 0000
5    0000000C 1234 5678 9ABC          DC.X    SYM2
              DEF0 0000 0000
6    00000018 1234 5678 9ABC    SYM3  DC.X    :1234_5678_9ABC_DEF_   ;hexadecimal representation
              DEF0 0000 0000
7                                                                    ; of floating-point numbers
8    00000024 1234 5678 9ABC    SYM4  DC.X    :123456789ABCDEF       ; using underscores
              DEF0 0000 0000
9    00000030 408F 0000 C8E3    SYM5  DC.X    35E42                  ;decimal representation
              E0BA AD7B 4E36
10   0000003C 408F 0000 C8E3    SYM6  DC.X    35_E_42                ; of floating-point numbers
              E0BA AD7B 4E36
11                                                                   ; using underscores
12                                    END
```

Figure 3-1. Floating-Point Constant Formats

A hexadecimal floating-point number is denoted by a colon (**:**) followed by a series of hexadecimal 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.

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 3-5.

Table 3-5. 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 long word operations, the assembler assigns values according to the following rules, which are compatible with the Motorola M68000 Family Resident Structured Assembler:

- In **DC** directives, character strings are always left-justified in words or long words. Any remaining bytes on the right of the word or long word are filled with zeros.

Examples:

```
DC.B 'A'          ; Hex value 41
DC.B 'AB'         ; Hex value 4142
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 41424344 45000000
```

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

Examples:

```
MOVE.B #'A',D0          ; Value moved is hex 41
MOVE.W #'A',D0          ; Value moved is hex 4100
MOVE.W #'AB',D0         ; Value moved is hex 4142
MOVE.L #'A',D0          ; Value moved is hex 00004100
MOVE.L #'AB',D0         ; Value moved is hex 00004142
MOVE.L #'ABC',D0        ; Value moved is hex 41424300
MOVE.L #'ABCD',D0       ; Value 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 3-6. 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*, in this manual.

The comparison operators =, >=, etc., return a logical true (all one bits) if the comparison is true and a logical false (zero) if the comparison is not true. All operands are considered to be unsigned 32-bit values and the comparison is unsigned. 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 not allowed in expressions. The assembler interprets spaces as termination characters.

Table 3-6. 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

(cont.)

Table 3-6. Summary of Operators and Their Precedence (cont.)

Precedence	Operator	Meaning
	-	Subtraction
7	=, <>	Equality, inequality
	>, >=	Greater than, greater than or equal
	<,<=	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:

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

.SIZEOF. — Specifies the Size of a Section

Syntax

`.SIZEOF. (`*section_name)*

Description

section_name	Specifies a section name.

The **.SIZEOF.** operator gives the size of the section *section_name*.

This operator gives a relocatable value. If an undefined section is referenced with a **.SIZEOF.** operator, the assembler will define a relocatable section with type **CODE** and alignment of 2 bytes. The value in **.SIZEOF.** is resolved at link time (with the default being 4 bytes).

Notes

See the related **.STARTOF.** operator.

Example

```
* This routine will clear memory for the full address
* range of the final combined section that contains
* the subsection zerovars from this module.
     OPT   CASE,D
     SECT  zerovars,,D
     DS.B  $300
     SECT  code,,C
     LEA   .STARTOF.(zerovars),A0 ; start of section
     MOVE  #.SIZEOF.(zerovars),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
```

.STARTOF. — Specifies the Starting Address of a Section

Syntax

`.STARTOF.` *(section_name)*

Description

section_name	Specifies a section name.

The **.STARTOF.** operator gives the starting address of the section *section_name*.

This operator gives a relocatable value. If an undefined section is referenced with a **.STARTOF.** operator, the assembler will define a relocatable section with type **CODE** and alignment of 2 bytes. A value in **.STARTOF.** is resolved at link time (with the default being 4 bytes).

Notes

See the related **.SIZEOF.** operator.

Instructions and Address Modes 4

Introduction

This chapter describes:

- The 68000/08/10/12/20/30/40/60, 68EC000/020/030/040/060, CPU32, 68851/881/882, and 68HC000/01 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 68000/08/10/12/20/30/40/60, 68360, 68HC000/01, 68EC000/020/030/040/060, 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 and 68060. The 68020 mnemonics, with the exception of **CALLM** and **RTM**, 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 behavior of the ASM68K Assembler 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 instructions:

```
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 4-1.

Table 4-1. 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

Examples:

```
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 that 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 in this chapter.

Note that there are minor differences in legal addressing modes between the CPU32 family, 68020/30/40/60, 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 4-2. Register mnemonics can be uppercase or lowercase and are reserved symbols.

Table 4-2. Summary of Register Mnemonics

Register	Description	Supported By
A0-A7	32-Bit Address Registers.	CPU32, 68000/10/20/30/40/60, 68EC000/20/30/40/60, 68HC000/01
AC	16-Bit Access Control Register.	68851
AC0-AC1	32-Bit Access Control Registers.	68EC030
ACUSR	Access Control Unit Status Registers.	68EC030
BAC0-BAC7	Eight 16-Bit Breakpoint Acknowledge Control Registers.	68851

(cont.)

Table 4-2. Summary of Register Mnemonics (cont.)

Register	Description	Supported By
BAD0-BAD7	Eight 16-Bit Breakpoint Acknowledge Data Registers.	68851
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.	68020/30/40/60, 68EC020/30/40/60
CAL	8-Bit Protection Control Register.	68851
CCR	Condition Code Register. **CCR** is the lower eight bits of the status register (**SR**).	CPU32, CPU32+, 68000/10/20/30/40/60, 68EC000/20/30/40/60, 68HC000/01
CONTROL/ FPCR	Floating-Point Control Register.	68040/60, 68881/68882
CRP	64-Bit User Root Point Register.	68030, 68851
D0-D7	32-Bit Data Registers.	CPU32, CPU32+, 68000/10/20/30/40/60, 68EC000/20/30/40/60, 68HC000/01
DFC/DFCR	Alternate Function Code Destination Register.	CPU32, CPU32+, 68EC010/20/30/40/60, 68010/20/30/40/60
DRP	64-Bit **DMA** Root Point Register.	68851
DTT0-1	Transparent Translation Registers.	68040/60
FP0-FP7	Floating-Point Data Registers.	68040, 68881/68882
IADDR/FPIAR	Floating-Point Instruction Address Register.	68040, 68881/68882

(cont.)

Table 4-2. Summary of Register Mnemonics (cont.)

Register	Description	Supported By
ISP	Interrupt Stack Pointer (68020/30 interrupt state).	CPU32, CPU32+, 68000/10/20/30/40/60, 68EC000/20/30/40/60, 68HC000/01
ITT0-1	Transparent Translation Registers.	68040/60
MMUSR	16-Bit MMU Status Register	68040/60
MSP	Master Stack Pointer (68020/30 supervisor state).	68020/30/40/60, 68EC020/30/40/60
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, CPU32+, 68000/10/20/30/40/60, 68EC000/20/30/40/60, 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, CPU32+, 68010/20/30/40/60
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, CPU32+, 68000/10/20/30/40/60
SRP	64-Bit Supervisor Root Point Register.	68030/40/60, 68851

(cont.)

Table 4-2. Summary of Register Mnemonics (cont.)

Register	Description	Supported By
SSP/A7	System Stack Pointer.	CPU32, CPU32+, 68000/10/20/30/40/60, 68EC000/20/30/40/60, 68HC000/01
STATUS/FPSR	Floating-Point Status Register.	68040/60, 68881/68882
TC	32-Bit Translation Control Register.	68030/40/60, 68851
TT0-TT1	32-Bit Transparent Translation Registers.	68030
URP	User Root Pointer Register.	68040/60
USP	User Stack Pointer (for user state).	CPU32, CPU32+, 68000/10/20/30/40/60, 68EC000/20/30/40/60, 68HC000/01
VAL	8-Bit Protection Control Register.	68851
VBR	Vector Base Register. Used for multiple vector table areas.	CPU32, CPU32+, 68010/20/30/40/60, 68EC020/30/40/60
ZA0-ZA7	Suppressed Address Registers. The register specified is used in the instruction, but its value is considered zero for effective address calculations.	68020
ZD0-ZD7	Suppressed Data Registers. The register specified is used in the instruction, but its value is considered zero for effective address calculations.	68020
ZPC	Suppressed Program Counter. The **PC** is used in the instruction, but its value is considered zero for effective address calculations.	68020

The 68881/882 floating-point coprocessor uses the 68020/30/40/60 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 4-3 lists the addressing modes that are defined for all chips. These modes are described in more detail in the following sections.

Table 4-3. Addressing Modes

	68000/08/ 1068302 68EC000 68HC000 68HC001	**CPU32 68330 68331 68332 68333 68340**	**68020/30/40/60 68EC020 68EC030 68EC040/060 68349 68360 CPU32+**
Register Direct Modes:			
Address Register Direct	yes	yes	yes
Data Register Direct	yes	yes	yes
Memory Addressing Modes:			
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 with16-bit Displacement	yes	yes	yes

(cont.)

Table 4-3. Addressing Modes (cont.)

	68000/08/ 1068302 68EC000 68HC000 68HC001	CPU32 68330 68331 68332 68333 68340	68020/30/40/60 68EC020 68EC030 68EC040/060 69349 68360 CPU32+
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
Memory Reference Modes:			
Absolute Short Address	yes	yes	yes
Absolute Long Address	yes	yes	yes
Program Counter Indirect with16-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
Non-Memory Reference Modes:			
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 68020/30/40/60 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 Data 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 long word (**.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 long word (**.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 16 bits sign-extended. On the CPU32 family and 68020/30/40/60, 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:

- From $FF8000 to $FFFFFF for the 68000/10
- From $F8000 to $FFFFF for the 68008
- From $FFFF8000 to $FFFFFFFF for the CPU32 family and 68020/30/40/60

Regardless of the target chip, the assembler recognizes only the absolute addresses from $FFFF8000 to $FFFFFFFF 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, 68060, 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, 68040, 68060, 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 16-bit displacement and 4 more for each 32-bit displacement.

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 16-bit displacement and 4 more for each 32-bit displacement.

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 vary 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 68020/30/40/60 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 68020/30/40/60 evaluates the instruction, transfers the operands through the coprocessor interface, and performs the addressing mode calculations.

Any of the 68020/30/40/60 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 68020/30/40/60 evaluates the instruction to its addressing modes. These include all legal 68020/30/40/60 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 4-4 provides a definition of terms used to describe operand syntax.

Table 4-4. Assembler Syntax for Effective Address Fields

Operand	Resultant Register or Expression
A*n*	Represents an address register.
D*n*	Represents a data register.
R*n*	Represents either an address or data register, or a suppressed register (**ZA***n* or **ZD***n*). ASM68K does not recognize the mnemonic R*n*.
abs_exp	Represents an absolute expression, including an external reference with no section specified.
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 68020/30/40/60 syntax. This syntax uses square brackets (**[ ]**) to designate the components of the intermediate address in the 68020/30/40/60 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):
  - Using a **Z**-register (**ZPC**, **ZA***n*, or **ZD***n*).
  - Using square brackets.
  - 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)
  - Specifying a scale factor other than 1 on an Index Register.
  - 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 **NOP**s will trail the instruction). The **OPT** flags **BRW** and **FRS** do not apply to forward references that appear in conjunction with a register.

Note that coding **(***exp***,A***n***)** rather than *exp***(A***n***)** or specifying a scale factor of 1 explicitly is not sufficient to force the use of 68020/30/40/60 modes.

Errors will occur when assembler syntax forces 68020/30/40/60 addressing modes, and the target microprocessor specified with the **CHIP** or **OPT P** directives is not the 68020, 68030, 68040, or 68060. You should note that:

- Assembler syntaxes that generate Address Register Indirect with Displacement or Memory Indirect Modes (e.g., **(***exp***,A***n***)** or **([***exp***,A***n***],R***n***)**) 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 **A***n* 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 68020/30/40/60 notation is used, but the list above should be kept in mind. For example, the discussion of the operand **(***abs_exp***,A***n***,R***n***{.W,.L})** includes the forms *abs_exp***(A***n***,R***n***{.W,.L})** and **(***abs_exp***,R***n***{.W,.L},A***n***)**.

Operand Syntax and Addressing Modes

The relationship between operand syntax and addressing modes is described below.

Dn Operand

The operand **D***n* always results in Data Register Direct Mode.

An Operand

The operand **A***n* always results in Address Register Direct Mode.

(An) Operand

The operand **(A***n***)** always results in Address Register Indirect Mode.

(An)+ Operand

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

-(An) Operand

The operand **-(A*n*)** always results 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***,A***n***)** 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,A***n***)** 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 **(D***n***)**, **(R***n***.W)**, **(R***n***.L)**, **(***abs_exp***,D***n***)**, **(***abs_exp***,R***n***.W)** and **(***abs_exp***,R***n***.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***,A***n***,R***n***{.W,.L})** and **(A***n***,R***n***{.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 that 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 that 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***,R***n***{.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 **A***n* and otherwise as the Index Register.

(rel_exp,An,Rn{.W,.L}) Operand

The operand **(***rel_exp***,A***n***,R***n***{.W,.L})** generates the Address Register Indirect with Base Displacement and Index Mode. The first **A** register without a size code or scale factor that 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***,D***n***,R***n***{.W,.L})** and **(D***n***,R***n***{.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:

- 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).
- 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.
- 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,R*n*{.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*,D*n*,R*n*{.W,.L})** and **(D*n*,R*n*{.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 Modes. 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 run time 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:

- These rules do not apply to the **B***cc* or **DB***cc* instructions, which always use Program Counter plus Displacement modes.
- 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 a **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, the Absolute Short Address Mode is used if the operand is in short-addressable memory and the 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:

- Relocatable sections or external references can be specified as short (see Chapter 4, *Relocation*, in this manual 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.
- 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.
- 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.
- The option flag **F** can be set with the **OPT** directive, causing all forward references except those in relative branch instructions (**B***cc*) 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 an **NOP**.

- The option flag **B** can be set with the **OPT** directive, which applies only to the relative branch instructions (**B***cc*) and causes forward references in one of these instructions to use the shorter form of the instruction, with 8-bit displacement. It is possible that sufficient room for the actual displacement will not exist, and unnecessary errors could occur.
- Individual **B***cc*, **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 **B***cc* 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 **B***cc***.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 **B***cc***.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 A*n*-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 that 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***(A***n***)** or **(***exp***,A***n***,R***n***)**, 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*(**A***n*) or (*rel_exp*,**A***n*)), 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:

effective_address = **A***n* + *displacement*
displacement = *effective_address* - **A***n*
displacement = *relocatable_expression* - **A***n*

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 that 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 4-1. 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    00000006 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 4-1. 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 that 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 run time, 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 4-2. The linker map file in Figure 4-3 shows the **INDEX** command used with an offset. The linker map file in Figure 4-4 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 4-2. A2-A5 Relative Addressing Example

```
Microtec Research ASM68K   Version x.y    Wed Jul 22 13:11:53 1992    Page  2


                   Symbol Table

Label          Value

?A2          External
VAR     DATA:00006000
```

Figure 4-2. 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=0FF0000H    ; Run-time value of A2 = 0FF0000H + 8000H,
                      ;                      = 0FF8000H.

* The displacement calculated for the "CLR VAR(A2)" instruction
* is the effective address of VAR (0FF0000 + 6000H) minus the
* run-time value of 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 4-3. Using the INDEX Command With Offset

```
Microtec Research LNK68K  Version x.y    Wed Jul 22 13:48:45 1992    Page   2


OUTPUT MODULE NAME:    TEST3
OUTPUT MODULE FORMAT:  IEEE


SECTION SUMMARY
---------------

SECTION    ATTRIBUTE                    START      END        LENGTH     ALIGN

PROG       NORMAL CODE                  00001000   00001009   0000000A   2 (WORD)
DATA       NORMAL DATA                  00FF0000   00FFFFFE   0000FFFF   2 (WORD)


MODULE SUMMARY
--------------

MODULE         SECTION:START        SECTION:END       FILE

TEST2             DATA:00FF0000        DATA:00FFFFFE  /test/TEST2.0
EST2.o
                  PROG:00001000        PROG:00001009



CROSS REFERENCE TABLE
---------------------

SYMBOL                                   SECTION          ADDRESS     MODULE

VAR                                      DATA             00FF6000   -TEST2
?A2                                                       00FF0000   -$$
                                                                      TEST2


START ADDRESS:   00000000

Load Completed
```

Figure 4-3. 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.sun4/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 4-4. Using the INDEX Command Without Offset

```
Microtec Research LNK68K  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   0000000A   2 (WORD)
DATA       NORMAL DATA                  00FF0000   00FFFFFE   0000FFFF   2 (WORD)


MODULE SUMMARY
--------------

MODULE         SECTION:START        SECTION:END       FILE

TEST2             DATA:00FF0000        DATA:00FFFFFE  /test/TEST2.0
EST2.o
                  PROG:00001000        PROG:00001009



CROSS REFERENCE TABLE
---------------------

SYMBOL                                    SECTION          ADDRESS     MODULE

VAR                                       DATA             00FF6000   -TEST2
?A2                                                        00FF0000   -$$
                                                                       TEST2


START ADDRESS:   00000000

Load Completed
```

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

Relocation 5

Introduction

Object modules produced by the ASM68K Assembler are in a relocatable format that lets you write modular programs whose final addresses will be resolved 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 and projects.
- The linker will resolve 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 sections is predefined to be 00 (numbered section 0). Each relocatable section has four attributes:

- Common/noncommon
- Short/long
- Section alignment
- Section type

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. 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 where the object module is included in the link.

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

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*, in this manual 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*, in this manual.

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 can 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 processor, such as the 68020, 68030, 68040, 68060, 680330, 680331, 680332, 680333, 680334, 68EC020, 68EC030, 68EC040, 68EC060, or CPU32.

A section alignment attribute of 4 combined with the **ALIGN 4** directive can ensure that data items are located at long word boundaries, which can 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 that contributes to the section is aligned modulo 4, thereby 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*, in this manual.

Section Type Attributes

There are four types of relocatable sections:

- C — Program code
- D — Data
- M — Mixed code and data
- 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*, in this manual.

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 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*, in this manual.

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 I, *HP 64000 Development System Support*.

When producing HP 64000 symbol files, the symbols from the various sections are mapped onto the HP 64000 sections (see Table 5-2).

Table 5-1. 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 5-2, there will be a conflict 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 a warning (604).
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 5-2 shows how a section is assigned the common/noncommon and short/long attributes.

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 appears for a second time in a **SECT** statement, it is then assigned the noncommon attribute as well, and a warning is produced.

Table 5-2. 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*S HORT	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 and external references. 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 that marks the symbol as absolute or relocatable. If relocatable, the type also indicates the section where 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.

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, 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.
- It has been declared 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.

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.

Expressions that produce a relocatable result are:

ABS+REL *REL+ABS* *REL-ABS* *REL+REL* *REL-REL*
*REL*ABS* *ABS*REL* *REL/ABS* *ABS/REL*

Expressions that produce an absolute result are:

REL=REL *REL<>REL* *REL<=REL*
REL<REL *REL>=REL* *REL>REL*
REL-REL *REL+REL*

The *REL-REL* and *REL+REL* expressions produce absolute results only if both *REL* subexpressions are in the same source program section and defined in the current module (no externals). Relocatable symbols that appear in expressions with any other operators will cause an error (e.g., *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 starting address of a relocatable section or references to the starting addresses of two or more relocatable sections.

For information on valid expressions, see the section *Expressions* in Chapter 3, *Assembly Language*, in this manual.

Assembler Directives 6

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.

Directive Summary

The notational conventions used to describe the syntax of assembler directives are the same as those presented in the *Preface* of this manual. The assembler directives in this chapter are organized alphabetically. Table 6-1 provides an alphabetical listing of the assembler directives.

Table 6-1. Alphabetical Listing of the Assembler Directives

Directive	Function
ALIGN	Specifies instruction alignment
CHIP	Specifies target microprocessor
COMLINE	Specifies memory space
COMMON	Specifies common section
DC	Defines constant value
DCB	Defines constant block
DS	Defines storage
ELSEC	Specifies conditional assembly converse
END	Ends assembly
ENDC	Ends conditional assembly code

(cont.)

Table 6-1. Alphabetical Listing of the Assembler Directives (cont.)

Directive	Function
ENDR	Ends repeat
EQU	Equates a symbol to an expression
FAIL	Generates programmed error
FEQU	Equates a symbol to a floating-point expression
FOPT	Specifies floating-point options
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 nonnegative (conditional assembly)
IFGT	Checks if value greater than zero (conditional assembly)
IFLE	Checks if value nonpositive (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	Checks if value unequal to zero (conditional assembly)
INCLUDE	Includes source file
IRP	Specifies indefinite repeat
IRPC	Specifies indefinite repeat character
LIST	Generates assembly listing
LLEN	Sets length of line in assembler listing
MASK2	Generates code to run on MASK2 (R9M) chip
MERGE_END	Indicates end of mergeable code or data
MERGE_START	Indicates start of mergeable code or data

(cont.)

Table 6-1. Alphabetical Listing of the Assembler Directives (cont.)

Directive	Function
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 n
```

Description

The **ALIGN** directive specifies the modulo number of bytes to which the address of the next instruction is to be aligned. *n* may be any power of 2 from 1 to 256.

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

type	Specifies target microprocessor. Valid values are: 68000, 68EC000, 68HC000, 68HC001, 68008, 68010, 68020, 68EC020, 68030, 68EC030, 68040, 68EC040, 68060, 68EC060, 68302, 68851, 68881, 68882, 68330, 68331, 68332, 68333, 68340, 68349, 68360, CPU32, and CPU32P. (default: **68000**)
cotype	Specifies target coprocessor. Valid values are 68881 and 68851. When using the 68851 coprocessor, the 68030, 68040, and 68060 processor types are illegal. When using the 68881 coprocessor with the 68040/60 processor, all floating-point instructions will be supported, including those not supported by the 68040/60 on-chip FPU. (default: **68881**)

The **CHIP** directive specifies the target microprocessor. Table 6-2 shows target processor values and the processor instruction set supported.

Table 6-2. Processor Identification

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

(cont.)

Table 6-2. Processor Identification (cont.)

Processor Value	Supported Instruction Set
68EC030	68EC030
68040	68040
68060	68060
68EC040	68EC040
68EC060	68EC060
68302	68000
68330	CPU32
68331	CPU32
68332	CPU32
68333	CPU32
68340	CPU32
68349	CPU32
68360	CPU32
68851	68851
68881	68881
68882	68882
CPU32	CPU32
CPU32P	CPU32

The differences in processor instruction sets are as follows:

- 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** directive is not set to 68010, a warning is generated, and the 68010 instruction is assembled.
- The 68020 carries the standard instruction set of the 68000 family and has the additional instructions **BFCHG**, **BFCLR**, **BFEXTS**, **BFEXTU**, **BFFFO**, **BFINS**, **BFSET**, **BFTST**, **BKPT**, **CALLM**, **CAS**, **CAS2**, **CHK2**, **CMP2**, **DIVSL**, **DIVUL**, **PACK**, **RTM**, **TDIVS**, **TDIVU**, **TRAP***cc*, **T***cc*, **TP***cc*, and **UNPK**. It has eight 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** directive is not set to 68020, 68030, 68040, or 68060 produces an error.

Using Motorola 68020, 68030, 68040, or 68060 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.

- 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**.
- The 68851 has the standard 68000 family instructions, as well as the additional instructions **PB***cc*, **PDB***cc*, **PFLUSH**, **PFLUSHA**, **PFLUSHS**, **PFLUSHR**, **PLOADR**, **PLOADW**, **PMOVE**, **PRESTORE**, **PSAVE**, **PS***cc*, **PTESTR**, **PTESTW**, **PTRAP***cc*, 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**.
- The CPU32 family has the same instructions as the 68010 with the following additions: **B***cc***.L**, **BGND**, **BRA.L**, **BSR.L**, **CHK.L**, **CHK2**, **CMP2**, **DIVS.L**, **DIVSL**, **DIVU.L**, **DIVU**, **EXTB**, **LINK.L**, **LISTOP**, **MULS.L**, **MULU.L**, **TBLS**, **TBLSN**, **TBLU**, **TBLUN**, **TRAP***cc*, **TST A***n*, and **TST #***<data>*.
- The 68040 has the same instruction set as the 68020 and 68030 with the following additions: **CINVL**, **CINVP**, **CINVA**, **CPUSHA**, **CPUSHL**, **CPUSHP**, **MOVE16**, **PFLUSHAN**, and **PFLUSHN**.
- The 68060 has the same instruction set as the 68040 with the following exceptions: the **PLPA** instruction is added for the 68060, and the **PTEST** instruction is not supported.

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

Note

Previous versions of the assembler allowed a symbol or expression to be used with the **CHIP** command. This option has been discontinued.

Example

```
CHIP 68020
```

This example specifies the 68020 processor.

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).

Notes

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

Example

```
COMLINE 8
```

This example reserves 8 bytes in memory.

COMMON — Specifies Common Section

Syntax

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

Description

label	Specifies the section name. If *sname* has been specified, *label* will be assigned the address of the current location counter (if it is a normal label).
`.S`	Specifies whether the section has the short attribute.
sname	The section name. Any valid section can be used.
snumber	The section number. One or two decimal digits can be used. If *label* is not specified, *snumber* is treated as the name of the common section. If *label* is specified, *label* is appended to *snumber* to create the name of a unique common section, such as **00***label*. This common section is not considered to have any particular connection with the noncommon (or common) numbered section 0.
n	The number of bytes of alignment. It can be 2 or 4. (default: 2)
type	The type of section. It can be:

`D`	Data
`R`	ROMable data

hptype	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.

Notes

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:

- There can be at most one **DATA** and one **COMN** section for each module.
- 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 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 Value

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`	Long word data
`.S`	Single precision floating
`.D`	Double precision floating
`.X`	Extended floating
`.P`	Packed binary coded decimal

(default: **.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 is evaluated to 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.

Notes

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. **DC** with any other qualifier will not accept floating-point numbers as operands.

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.

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 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.

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.

```
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.W10,5,7
00FF             DC$FF
3F80 0000        DC.S1.0
3FF0 0000 0000   DC.D1.0
0000
3F80 0000        DC.S:3F8
3FF0 0000 0000   DC.D:3FF
0000
```

This example shows the assembler directives with operands and the bytes generated (on the left). Note that spaces are not allowed between multiple operands that are separated by commas.

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`	Long words
`.S`	Single precision
`.D`	Double precision
`.X`	Extended floating
`.P`	Packed binary coded decimal

(default: **.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, long words, 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,$FFFFFFFF
```

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`	Long words
`.S`	Single precision
`.D`	Double precision
`.X`	Extended floating
`.P`	Packed binary coded decimal

(default: **.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, long words, 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.

Examples

```
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 comment field 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
                                                    ;Buffer start
18 0000000C 1219              MAIN10 MOVE.B  (A1)+,D1
19 0000000E 0C01 0014               CMP.B   #BLNK,D1
                                                ;Check for nonblank
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 that is subject to the conditional assembly statement ends. In the case of nested **IF***xx* statements, an **ENDC** is paired with the most recent **IF***xx* statement.

Example

```
MOVE    #22,D2
IFNE    SUM-4
ORI     #200,D3      ; assembled if
ADD     D0,VALUE+3   ; SUM-4 is nonzero
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 `IFNE` 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**, **IRP**, or **IRPC** directives.

Notes

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

label	Defines symbol name.
expression	Sets the symbol to the value of *expression* for the duration of the current assembly. Any attempt to reequate the same label will result in an error. Any symbol used in *expression* must have been previously defined, externally defined, or defined as a simple forward reference. If *expression* uses previously defined symbols, the standard expression rules apply (see the section *Expressions* in Chapter 3, *Assembly Language*, in this manual). If *expression* uses an externally defined symbol, a constant number can be added or subtracted from the symbol. If *expression* is defined as a forward reference, only one symbol can be in the *expression*, and no operators are allowed.
keyword	Sets the symbol to the value of *keyword*, 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
sym3    EQU      *           ; current program counter location
```

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. If the value of *expression* is 500 or more, a warning is generated; otherwise, an error message is generated. (default: **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.

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 (default)
`.P`	Packed decimal

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.

Notes

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 — Specifies Floating-Point Options

Syntax

`FOPT ID=`*n*

Description

`ID=`*n*	Specifies the coprocessor ID field. The range of *n* is 0 through 7. (default: **1**)

The **FOPT** directive specifies the coprocessor ID field (0 through 7) used in subsequent 68881/882 floating-point instructions.

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 that 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.

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).

Notes

The **IDNT** directive 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** or **NAME** directive should appear in a program.

Example

```
LAB1 IDNT
```

IFC — Checks if Strings Equal (Conditional Assembly)

Syntax

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

Description

string1	Represents a string of characters (see rules below).
string2	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 apply to *string1* (note that the term nonblank excludes tab characters):

- 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 here as the delimiting comma.

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

- 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, the second string extends 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.

Notes

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 in a macro
IFC   \1,\2                ; Parameters are expanded in a macro
IFC   string  , string     ; Equal (blanks not significant)
```

IFDEF — Checks if Symbol Defined (Conditional Assembly)

Syntax

`IFDEF` *symbol*

Description

symbol	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.

Notes

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.

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 +$7FFFFFFF. In contrast, the logical operator = performs unsigned comparisons, treating operands as 32-bit unsigned integers ranging from 0 to +$FFFFFFFF. Therefore, **IFEQ** *x* is not equivalent to **IFNE** *x*=**0**. Logical operators return a value of $FFFFFFFF for true and zero for false. For information on expressions, see the section *Expressions* in Chapter 3, *Assembly Language*, in this manual.

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                                    ELSEC
8                                       NOP    ; unassembled
9                                       NOP    ; unassembled
10                                   ENDC
11                                   IFEQ ONE
12                                      NOP    ; unassembled
13                                      NOP    ; unassembled
14                                   ELSEC
15   00000004 4E71                      NOP    ; assembled
16   00000006 4E71                      NOP    ; assembled
17                                   ENDC
18                                   END
```

IFGE — Checks if Value Nonnegative (Conditional Assembly)

Syntax

`IFGE` *expression*

Description

expression	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.

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 +$7FFFFFFF. In contrast, the logical operator **>=** performs unsigned comparisons, treating operands as 32-bit unsigned integers ranging from 0 to +$FFFFFFFF. Therefore, **IFGE** *x* is not equivalent to **IFNE** *x***>=0**. Logical operators return a value of $FFFFFFFF for true and zero for false. For information on expressions, see the section *Expressions* in Chapter 3, *Assembly Language*, in this manual.

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.

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 +$7FFFFFFF. In contrast, the logical operator > performs unsigned comparisons, treating operands as 32-bit unsigned integers ranging from 0 to +$FFFFFFFF. Therefore, **IFGT** *x* is not equivalent to **IFNE** *x*>**0**. Logical operators return a value of $FFFFFFFF for true and zero for false. For information on expressions, see the section *Expressions* in Chapter 3, *Assembly Language*, in this manual.

IFLE — Checks if Value Nonpositive (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.

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 +$7FFFFFFF. In contrast, the logical operator **<=** performs unsigned comparisons, treating operands as 32-bit unsigned integers ranging from 0 to +$FFFFFFFF. Therefore, **IFLE** *x* is not equivalent to **IFNE** *x***<=0**. Logical operators return a value of $FFFFFFFF for true and zero for false. For information on expressions, see the section *Expressions* in Chapter 3, *Assembly Language*, in this manual.

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.

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 +$7FFFFFFF. In contrast, the logical operator <= performs unsigned comparisons, treating operands as 32-bit unsigned integers ranging from 0 to +$FFFFFFFF. Therefore, **IFLT** *x* is not equivalent to **IFNE x<=0**. Logical operators return a value of $FFFFFFFF for true and zero for false. For information on expressions, see the section *Expressions* in Chapter 3, *Assembly Language*, in this manual.

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 **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 apply to *string1* (note that the term nonblank excludes tab characters):

- 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 here as the delimiting comma.

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

- 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, the second string extends 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.

Notes

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: assembly 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.

Notes

No forward reference is allowed.

Example

```
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 **ENDC** 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 **ENDC** directive. When *expression* is not 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 $FFFFFFFF for true and zero for false. For information on expressions, see the section *Expressions* in Chapter 3, *Assembly Language*, in this manual.

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** directories. If the search is still not successful, the assembler will search a list of directories specified in the **MRI_68K_INC** environment variable if it is set. For information on the correct syntax, see the section *Environment Variables* in Chapter 2, *UNIX/DOS User's Guide.*

Example

```
INCLUDE EXTERNAL.SRC
```

IRP — Specifies Indefinite Repeat

Syntax

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

Description

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

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.*

Notes

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 as follows:

```
Line Address
1                                   OPT  M
2                                   XREF SUB1,SUB2,SUB3
3
4                               ; Three JSR instructions
5                               ; are generated.
6                                   IRP  DUMMY,SUB1,SUB2,SUB3
7                                   JSR  DUMMY
8
9                                   ENDR
9.1  00000000 4EB9 0000 0000  E     JSR  SUB1
9.2
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

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

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.*

Notes

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 turns on listing output if a listing file has been requested with the **-l** option. The assembly listing is printed by default (**LIST**). When **NOLIST** is used, an assembly listing is not printed.

The **LIST** directive is synonymous with the **OPT S** 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.

MERGE_END — Indicates End of Mergeable Code or Data

Syntax

```
MERGE_END
```

Description

This directive indicates the end of a mergeable block of code or data. There must be a corresponding **MERGE_START** directive. This directive can only be used within relocatable sections.

Notes

This directive is used for Microtec compiler support. Use of it in manually written code is not recommended.

MERGE_START — Indicates Start of Mergeable Code or Data

Syntax

MERGE_START *symbol* , [G | P] , [N | X]

Description

This directive indicates the start of a mergeable block of code or data. There must be a corresponding **MERGE_END** directive. This directive may only be used within relocatable sections. It is expected that the symbol used with this directive exists within the mergeable block.

References to symbols within a mergeable block will use the largest address possible for the particular instruction that uses the reference. This is necessary since the actual address of the text may be in a different part of the section or possibly even a different section.

Another behavioral change is that program counter relative offsets will require relocation entries if there is a mergeable block between the source and the destination or if the destination is within a mergeable block.

Notes

This directive is used for Microtec compiler support. Use of it in manually written code is not recommended.

Examples

```
        MERGE_START _f1,G,X
_f1:
        . . .
        MERGE_END
```

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 **NOOBJ** directive is synonymous with **OPT -O**.

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.

Examples

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 |W
|X | ABSPCADD | BRB | BRL | BRS | BRW | CASE | CEX | CL | CRE | FRL
| FRS | MC | MD | MEX | NEST=n | OLD | OP=n | P=chip[/cotype] | PCO
| PCR | PCS | QUICK | REL32]
```

Description

The many options are listed here alphabetically.

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 PC-relative mode, rather than 5+*current_PC*. 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 the section *Addressing Mode Syntax* in Chapter 4, *Instructions and Address Modes*, in this manual. (default: **ABSPCADD**)
B BRB BRS	Forces forward references in relative branch instructions (**B***cc*, **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 (**B***cc*, **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 (**B***cc*, **BRA**, **BSR**) that have forward ref-

	erences. **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, **LOOP5** is different from **loop5**. 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 errors only, 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 **B***xx* 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=`*n*	Sets the nesting levels for macros. The default is set to the maximum number of nesting levels. (default: **NEST=100** for UNIX or **NEST=20** for 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 **B***cc* 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 (68020/30/40/60, 68EC020, 68EC030, 68EC040, 68EC060, and CPU32 family). (default: **OLD**)
`OP=`*n*	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 68851 coprocessor is compatible with all 68000 family processors with the exception of 68030, 68040, and 68060.

Example

```
        P=68020
```

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

The preceding `NO` or minus sign is not permitted with the `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 intra-section 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*, **D***n*	MOVEQ	*#data*, **D***n*
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 that 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 always had to size cast displacement expressions if you had a processor that accessed a full

	32-bit address range (68020/30/40/60 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, 68060, or CPU32 family. (default: **NOREL32**)
`S`	Turns on listing output if a listing file has been requested with the **-l** option. 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 an Absolute Section

Syntax

`ORG[.`*qualifier*`] [`*expression*`] [* {+ | -}` *displacement*`] [,`*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).
`* {+ | -}` *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.

Notes

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. (default: **60**)

PLEN specifies the number of lines on an assembly listing page.

Example

```
PLEN 58
```

REG — Defines a 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 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

label	Assigns the address of the current program counter to *label*.
count	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.
sname	Specifies the noncommon section name. Any valid section can be used.
snumber	Specifies the section number. Up to two decimal digits can be used.
align	Specifies the bytes of alignment, which can be 0, 1, 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 5, *Relocation*, in this manual.
type	Specifies the type of section. It can be:

`C`	Code
`D`	Data
`M`	Mixed code, data, etc.
`R`	ROMable data

hptype	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 the section *Section Type Attributes* in Chapter 5, *Relocation*, in this manual.

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:

- There can be at most one **PROG**, one **DATA**, and one **COMN** section for each module.
- 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.

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 I, *HP 64000 Development System Support,* in this manual.

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

label	Specifies symbol name.
expression	Assigns value to *label* 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.

A **SET** directive is not allowed to be forward referenced (referred to before it is actually defined). A symbol defined with **SET** is limited in scope. A symbol's current value under **SET** is visible from the point of definition until another **SET** directive redefines that symbol, or the end of the assembly file is reached. An example incorrect usage of **SET**:

```
DC.W    TITI
??0003  EQU     *
TITI    SET     *-??0003
        END
```

The assembler gives an undefined symbol error for the statement `DC.W TITI`. Use the **EQU** directive to define symbols that are visible throughout the assembly file; changing the **SET** directive to an **EQU** in the third line would assemble the code correctly.

Like **EQU**, this directive can also be used to define new keywords.

Examples

```
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 and is case-sensitive. Additional characters are ignored.
	You can optionally delimit the heading with single quotes ('). 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 until link time.

XCOM is a Microtec 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.

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***(A***n***)**).

Notes

Section and short/long specification are not supported.

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.

Notes

Specifying the section name (or number) of an external reference sometimes affects the assembler's choice of addressing mode (see Chapter 4, *Instructions and Address Modes*, in this manual). 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***(A***n***)**).

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, see the section *Program Sections* in Chapter 5, *Relocation*, in this manual.

Example

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

Macros 7

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.

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.

Using brackets is 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 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 equals 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.

Examples:

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 and by type. An alphabetical listing of these directives is shown in Tables 7-1 and 7-2.

Table 7-1. Assembler Macro Directives

Macro Directive	Function
ENDM	Terminates macro definition
LOCAL	Defines local symbol
MACRO	Enters macro definition
MEXIT	Exits macro

Table 7-2. Assembler Preprocessor Directives

Directive	Description
#define *symbol* #define *symbol expression*	Defines a symbol
#if *expression* [#else] #endif	Specifies conditional compilation if expression is TRUE
#ifdef *symbol* [#else] #endif [*symbol*]	Specifies conditional compilation if symbol is defined
#ifndef *symbol* [#else] #endif [*symbol*]	Specifies conditional compilation if symbol is not defined.

#include <*file*>
#include "*file*"
	Includes file

#undef *sym*	Removes a symbol definition

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 a 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         LAB1
LAB2\@        MOVE.B        #TIME,D0
LAB1          DBLE          D0,LAB2\@
              ENDM

+??0002       MOVE.B        #5,D0         ;first call
+??0001       DBLE          D0,??0002     ;with TIME=5
+??0004       MOVE.B        $FF,D0        ;second call
+??0003       DBLE          D0,??0004     ;with TIME=$FF
```

In this example, 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

label	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.
parameter	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 an **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 3200LXX  MOVE   D0,D1   ; Create label using parameter
6      END
```

MEXIT — Exits Macro

Syntax

[*label*] `MEXIT`

Description

label	Assigns the address of the current location counter to *label*.

The **MEXIT** directive is an alternate method for terminating a macro expansion. During a macro expansion, a **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 this example, the code following **MEXIT** will not be assembled if `DATA` is not zero.

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

#define — Defines a Symbol

Syntax

`#define` *symbol* `[` *expression* `]`

Description

The **#define** directive defines a symbol. If an *expression* is defined, the expression is bound to the macro *symbol* and is subsituted by the assembler whenever *symbol* is encountered in the code.

This macro simulates the function the C preprocessor macro of the same name.

#if — Compiles If Expression True

Syntax

```
#if expression
[#else]
#endif
```

Description

The **#if** directive specifies a conditional compilation if *expression* evaluates to **TRUE**. All statements up to the matching **#endif** directive are executed. If *expression* evaluates to **FALSE**, all statements up to the matching **#endif** directive are skipped.

The optional **#else** directive modifies the behavior of the **#if** directive by either executing all commands between the **#if** and **#else** if *expression* is **TRUE**, or all commands between the **#else** and the **#endif** if *expression* is **FALSE**.

This macro simulates the function the C preprocessor macro of the same name.

#ifdef — Compiles If Symbol Is Defined

Syntax

```
#ifdef symbol
[#else]
#endif [symbol]
```

Description

The **#ifdef** directive specifies a conditional compilation of code if *symbol* has been defined to the assembler. All assembly code between the **#ifdef** statement and the matching **#endif** statement is executed if symbol is defined; otherwise, the code is skipped.

The optional **#else** directive modifies the behavior of the **#ifdef** directive by either executing all commands between the **#ifdef** and **#else** directives if *symbol* is defined, or between the **#else** and **#endif** if *symbol* is not defined.

This macro simulates the function the C preprocessor macro of the same name.

#ifndef — Compiles If Symbol Is Not Defined

Syntax

```
#ifndef symbol
[#else]
#endif [symbol]
```

Description

The **#ifndef** directive specifies a conditional compilation of code if *symbol* has not been defined to the assembler. All assembly code between the **#ifndef** statement and the matching **#endif** statement is executed if *symbol* is not defined; otherwise, the code is skipped.

The optional **#else** directive modifies the behavior of the **#ifndef** directive by either executing all commands between the **#ifndef** and **#else** directives if *symbol* is not defined, or between the **#else** and **#endif** if *symbol* is defined.

This macro simulates the function the C preprocessor macro of the same name.

#include — Includes a Source File

Syntax

```
#include <file>
#include "file"
```

Description

The **#include** directive includes *file* as part of the assembler code. All functions and variables defined in *file* can be accessed as if they were local to the assembler file.

If *file* is enclosed by angle brackets (<>), the preprocessor will look in the stamdard **#include** directories for *file*.

If *file* is enclosed by quotes (`" "`), the preprocessor will look in the current directory first, followed by the standard **#include** directories.

This macro simulates the function the C preprocessor macro of the same name.

#undef — Removes a Symbol Definition

Syntax

#undef symbol

Description

The **#undef** directive removes a symbol definition previously defined with a **#define** *symbol* directive.

This macro simulates the function the C preprocessor macro of the same name.

Structured Control Directives 8

Introduction

ASM68K includes several high-level language constructs similar to C constructs that 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 requires 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:

- A condition code (**CC**, **EQ**, etc.) enclosed in angle brackets (e.g., **<MI>**).

  This type of structured control expression generates a conditional branch instruction (**B***cc*), which tests the indicated bits of the condition codes. Any of the 14 condition codes accepted in the conditional branch instruction (**B***cc*) are 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.

- Two expressions (as defined in Chapter 3, *Assembly Language*, in this manual), 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 pound 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.

- 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 5, *Relocation*, in this manual. 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:

- 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.
- 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**.
- A compound expression containing **AND** is true if the expressions on both sides of **AND** are true; otherwise it is false.
- A compound expression containing **OR** is false 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**. 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 8-1.

Table 8-1. Conditional Code Comparisons

Condition Code Conversion			New Condition
<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 listing when they are not equivalent.
<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 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                                   FOR.B D1 = #1 TO #10 DO.S
4.1  00000000 123C 0001               MOVE.B #1,D1 ;>> FOR <<
4.2  00000004 6002                    BRA.S ??0001 ;>> FOR <<
4.3  00000006 5201           ??0002  ADD.B #1,D1  ;>> FOR <<
4.4  00000008 0C01 000A      ??0001  CMP.B #10,D1 ;>> FOR <<
4.5  0000000C 6E04                    BGT.S  ??0004 ;>> FOR <<
5    0000000E 34C1                   MOVE.W D1,(A2)+
6                                    ENDF
6.1  00000010 60F4           ??0003 BRA ??0002 ;>> ENDF <<
6.2                          ??0004 ;>> ENDF <<
7                                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 **SET**s only affect variable values at assembly time.

Structured Control Directives

Each of the structured control directives generates one or more assembly language instruction, which typically includes compare and branch instructions. These statements are used to control looping or conditional execution at run time, not at assembly time. Table 8-2 lists the structured control directives.

Table 8-2. Alphabetical Listing of the Structured Control Directives

Directive	Function
BREAK[a]	Exits loop prematurely
FOR . . . ENDF[b]	Specifies FOR loop
IF . . . ELSE . . . ENDI[b]	Specifies IF-ELSE construct
NEXT[a]	Proceeds to next iteration of loop
REPEAT . . . UNTIL[b]	Specifies REPEAT loop
WHILE . . . ENDW[b]	Specifies WHILE loop

a. Microtec has extended the Motorola control directives to add two directives that alter the flow of loop constructs.

b. 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 run time. 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 — Specifies FOR Loop

Syntax

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

Description

qualifier	Applies this size qualifier to **MOVE**, **CMP**, **ADD**, or **SUB** instructions within the loop. Valid values for *qualifier* are: **B**, **W**, or **L**.
op1	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).
op2	Specifies initial value of the loop counter.
op3	Specifies final value of the loop counter.
op4	Increments (for TO) or decrements (for DOWNTO) loop counter by this value. If *op4* is not specified, the loop is incremented or decremented by 1.
extent	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**).
loop_body	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 DOWNTO), 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 pound (**#**) 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 — Specifies IF-ELSE Construct

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 IFs.
; As you see, the combination is sometimes difficult to understand.

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 IF is illegal
 IF VARIABLE <EQ> #0 THEN.S     ; structured IF
  MOVE #1,VARIABLE
ENDC                            ; conditional
ENDI                            ; structured IF - terminates
                                ; whichever of the preceding
                                ; structured IFs was assembled
```

NEXT — Proceeds to Next Iteration of Loop

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 — Specifies Repeat Loop

Syntax

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

Description

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.
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.

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.

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 — Specifies WHILE Loop

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**).

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.

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.

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
```

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

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
   RTS                              ; Jump here on BREAK
```

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 9

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 9-1. 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, the section in which they were defined, and 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*, in this manual.

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 in Figure 8-1. Refer to the following points in order to examine and understand the cross-reference table format.

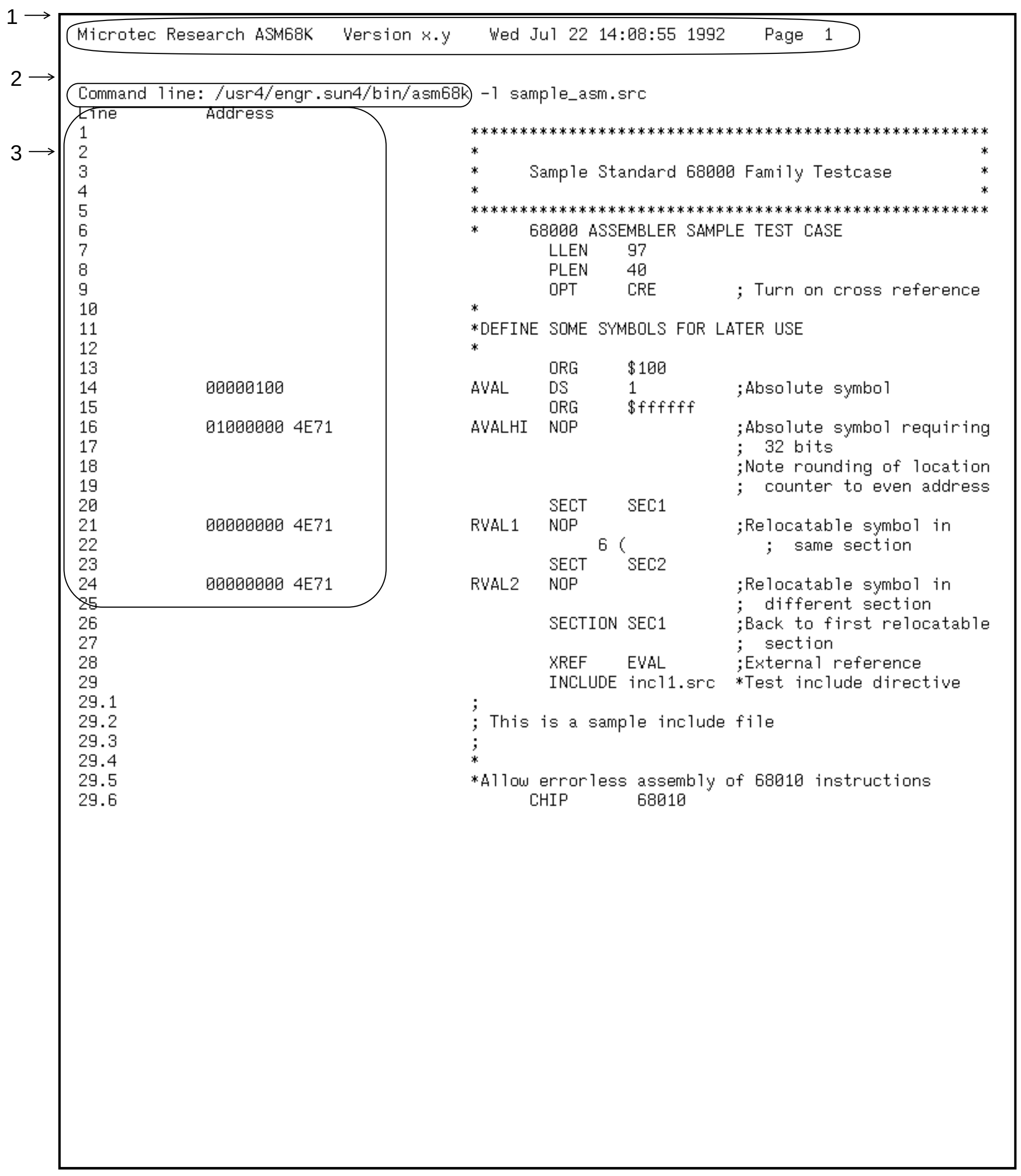
8. All user-defined symbols in the program are listed under the heading `Label`.

9. The symbol values are listed under `Value`. Any flag to the left of the values indicates 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 multiple defined symbols, more than one definition can appear for the symbol.

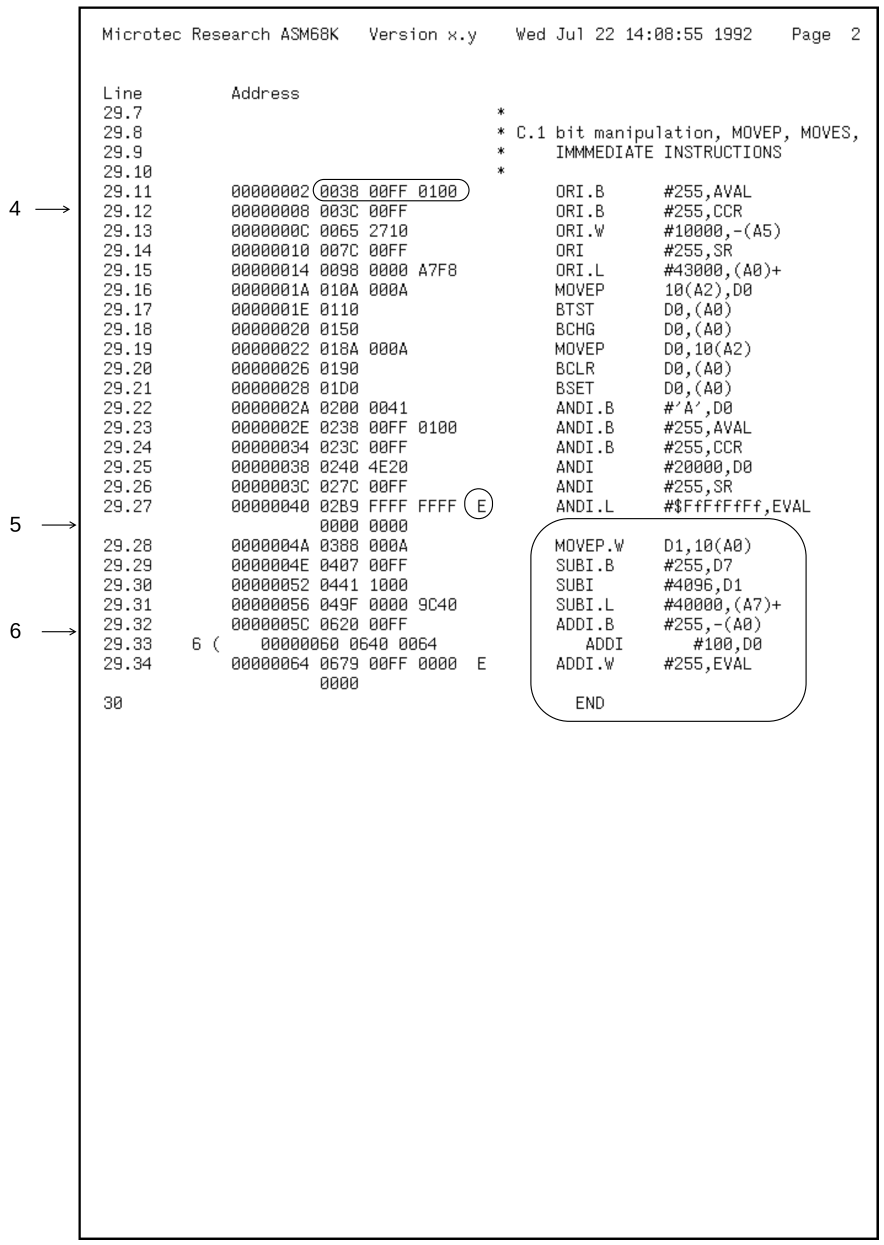
Section names, macro names, and the module name do not appear in the symbol table listing.

Sample Program Listing



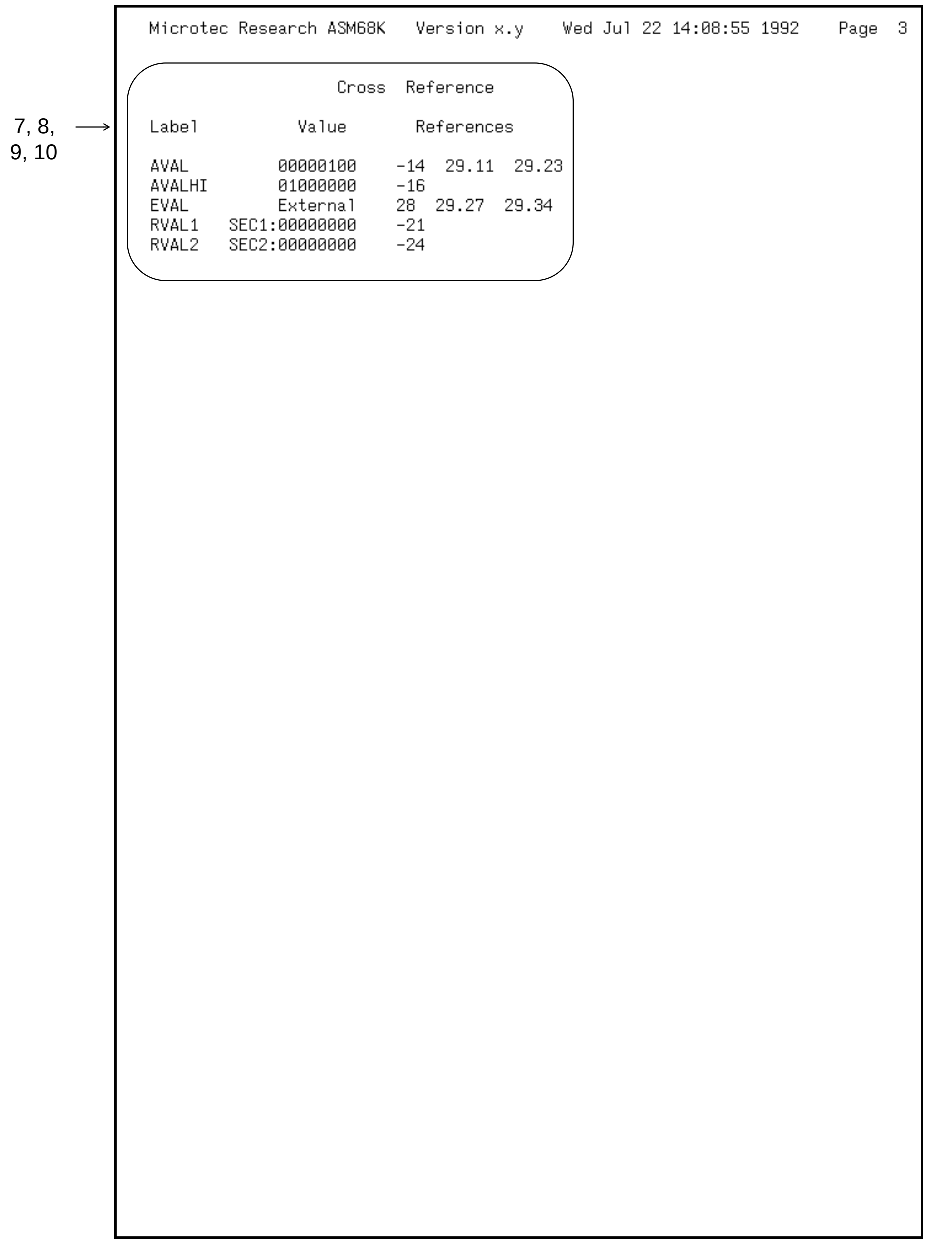
```
Microtec Research ASM68K   Version x.y    Wed Jul 22 14:08:55 1992    Page  1

Command line: /usr4/engr.sun4/bin/asm68k -l sample_asm.src
Line          Address
1                                        **************************************************
2                                        *                                                *
3                                        *     Sample Standard 68000 Family Testcase      *
4                                        *                                                *
5                                        **************************************************
6                                        *     68000 ASSEMBLER SAMPLE TEST CASE
7                                                LLEN    97
8                                                PLEN    40
9                                                OPT     CRE        ; Turn on cross reference
10                                       *
11                                       *DEFINE SOME SYMBOLS FOR LATER USE
12                                       *
13                                               ORG     $100
14            00000100                   AVAL    DS      1          ;Absolute symbol
15                                               ORG     $ffffff
16            01000000 4E71              AVALHI  NOP                ;Absolute symbol requiring
17                                                                  ;  32 bits
18                                                                  ;Note rounding of location
19                                                                  ;  counter to even address
20                                               SECT    SEC1
21            00000000 4E71              RVAL1   NOP                ;Relocatable symbol in
22                                                    6 (              ;  same section
23                                               SECT    SEC2
24            00000000 4E71              RVAL2   NOP                ;Relocatable symbol in
25                                                                  ;  different section
26                                               SECTION SEC1       ;Back to first relocatable
27                                                                  ;  section
28                                               XREF    EVAL       ;External reference
29                                               INCLUDE incl1.src  *Test include directive
29.1                                     ;
29.2                                     ; This is a sample include file
29.3                                     ;
29.4                                     *
29.5                                     *Allow errorless assembly of 68010 instructions
29.6                                           CHIP      68010
```

Figure 9-1. Sample Assembly Listing



```
Microtec Research ASM68K   Version x.y    Wed Jul 22 14:08:55 1992    Page  2


Line         Address
29.7                                         *
29.8                                         * C.1 bit manipulation, MOVEP, MOVES,
29.9                                         *     IMMMEDIATE INSTRUCTIONS
29.10                                        *
29.11        00000002 0038 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        #255,SR
29.15        00000014 0098 0000 A7F8             ORI.L      #43000,(A0)+
29.16        0000001A 010A 000A                  MOVEP      10(A2),D0
29.17        0000001E 0110                       BTST       D0,(A0)
29.18        00000020 0150                       BCHG       D0,(A0)
29.19        00000022 018A 000A                  MOVEP      D0,10(A2)
29.20        00000026 0190                       BCLR       D0,(A0)
29.21        00000028 01D0                       BSET       D0,(A0)
29.22        0000002A 0200 0041                  ANDI.B     #'A',D0
29.23        0000002E 0238 00FF 0100             ANDI.B     #255,AVAL
29.24        00000034 023C 00FF                  ANDI.B     #255,CCR
29.25        00000038 0240 4E20                  ANDI       #20000,D0
29.26        0000003C 027C 00FF                  ANDI       #255,SR
29.27        00000040 02B9 FFFF FFFF  E          ANDI.L     #$FfFfFfFf,EVAL
                      0000 0000
29.28        0000004A 0388 000A                  MOVEP.W    D1,10(A0)
29.29        0000004E 0407 00FF                  SUBI.B     #255,D7
29.30        00000052 0441 1000                  SUBI       #4096,D1
29.31        00000056 049F 0000 9C40             SUBI.L     #40000,(A7)+
29.32        0000005C 0620 00FF                  ADDI.B     #255,-(A0)
29.33    6 (    00000060 0640 0064                  ADDI       #100,D0
29.34        00000064 0679 00FF 0000  E          ADDI.W     #255,EVAL
                      0000
30                                                 END
```

Figure 9-1. Sample Assembly Listing (cont.)



```
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 9-1. 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 10

Overview

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:

- Resolve external references between modules and check for undefined references (linking)
- Adjust all relocatable addresses to the proper absolute addresses (loading)
- 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 IEEE-695 format, Motorola S-record format, or Hewlett-Packard HP-OMF format.

The LNK68K Linker 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 IEEE-695 format, S-record format, or HP-OMF format.
- Independently specified relocatable section load addresses.
- Specified relocatable section loading order.

- Incremental linking.
- Definition of external symbols at link time.
- Values of previously defined externals changed at link time.
- Loading of object modules from a library. The linker will include only those modules from a library that are necessary to resolve external references. Library modules can also be incrementally linked.
- Inclusion of symbols and line number information in the absolute object module for symbolic debugging.
- Cross-reference table generation.
- For better memory utilization, first-fit algorithm places object modules in memory.
- Complex relocation.
- Initializing data from ROM and other locations.
- Automatic copying of initialized data values, which can be placed in ROM.
- Scatter loading.
- A2-A5 address register-relative addressing.
- Removal of duplicate C++ template functions or data. The Microtec C++ compiler emits an instance of each C++ template function or template static data into each module that uses it. This can result in multiple copies of a given instance in many or all of the modules that are to be linked together. Only the first copy of a given C++ template function or template static data read by the linker will be retained.
- Removal of duplicate non-inline copies of in-line functions. If multiple non-inline copies of the same in-line function exist in more than one module, only the first copy read by the linker will be retained.

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 that 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

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 differs from an absolute section in that its input object modules do not specify an absolute load address before being input to the linker. The linker loads a relocatable section into a contiguous block of memory that does not overlap 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 that are associated with the section name. On occasion, it will be necessary to refer to the individual pieces of code from various modules that 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 Chapter 5, *Relocation*, in this manual. From the linker's point of view, all sections are named.

Each relocatable section has four attributes:

- Common/noncommon
- Short/long
- Section alignment
- Section type

Sections containing these types of attributes are described below. For more information on relocatable modules using these attributes, see Chapter 5, *Relocation*, in this manual.

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 that 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 that can be reached by a 16-bit sign-extended address (see Table 10-1). 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 an **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 that is not short and that can be placed anywhere in memory.

Table 10-1. 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 $FFFFF
68020/30/40/60, CPU32, 680330/331/332/333/340/360, 68EC020/030/040/060	0 to $7FFF $FFFF8000 to $FFFFFFFF

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 5, *Relocation*, in this manual.

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 in Chapter 11, *Linker Commands*, in this manual.

Section Type

A section can have one of the types listed in Table 10-2.

Table 10-2. 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. 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 also 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:

1. Absolute sections and sections specified by the **SECT** and **COMMON** linker commands

2. Short sections

3. Long sections

The order in which short or long sections are assigned memory is:

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 those commands

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 that 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 that 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 was 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

This subsection provides information about program identifiers.

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:

- The first character must be one of the following:

  a-z A-Z ? @ _ $

- Subsequent characters may be any of the above, plus:

  0-9

- The maximum symbol length is 512 characters. Symbols are truncated to 15 characters (on output, not internally) if **FORMAT HP** applies. 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 this example, `NEWNAME` and `newname` refer to different identifiers. The linker command **PUBLIC** is recognized in both upper- and lower-case letters.

Names originating from the Microtec 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 a pound sign without using the escape character (backquote (`)).

Linker Commands 11

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 Chapter 2, *UNIX/DOS 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 11-1 shows how numeric command arguments can be represented.

Table 11-1. Representation of Numeric Command Arguments

Type	Indicator
Hexadecimal	Preceded by **$** or terminated with **H**
Binary	Terminated with **B**
Octal	Terminated with **O**
Decimal	Terminated 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 is:

```
   . . .
   . . .
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

Symbols and section names must follow the syntax rules for symbols given in *Assembler Syntax* in Chapter 3, *Assembly Language*, in this manual.

Case Sensitivity

Section names, symbols, and reserved names (e.g., IEEE) are by default case-sensitive in the linker command file. The **CASE**, **LOWERCASE**, or **UPPERCASE** linker commands can alter this default.

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 **INCLUDE** are expanded before any linker commands are processed.
2. [**NO**]**INTFILE** must be placed before any **LOAD** command.
3. Position-independent commands are processed next:

   ALIGN
   ALIGNMOD
   BASE
   CHIP
   DEFINE
   ERROR
   FORMAT
   INDEX
   LISTABS
   LISTMAP
   NAME
   RESADD
   RESMEM
   SECTSIZE
   START
   WARN

4. Position-dependent commands are processed next:

CASE	(place before any command using names)
LOWERCASE	(place before any command using names)
UPPERCASE	(place before any command using names)
[NO]PAGE	
CPAGE	
[NO]DEBUG_SYMBOLS	
LOAD	
LOAD_SYMBOLS	
EXTERN	
PUBLIC	

5. Commands that are position-independent in the command file are processed next, but they are operated on in the following order:

   a. SECT, PUBLIC, COMMON

   b. MERGE

   c. ALIAS

   d. ORDER, SORDER

   e. 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 11-2. The commands are described in the pages following the table.

Table 11-2. 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
DEFINE	Defines a symbolic constant for linker commands
END	Ends command stream and finishes linking
ERROR	Modifies message severity
EXIT	Exits linker without linking
EXTERN	Creates external references

(cont.)

Table 11-2. Linker Commands (cont.)

Command	Function
FORMAT	Selects output format
INCLUDE	Includes a command file
INDEX	Specifies run-time value of register A*n*
INITDATA	Creates ROM section for RAM initialization
INTFILE	Stores information using intermediate file or virtual memory
LISTABS	Lists symbols to output object module
LISTMAP	Specifies layout and content of the map
LOAD	Loads specified object 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 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

sname	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.

Note

The linker will always generate the section **??INITDATA**. This section will either be empty or contain data depending on whether the **INITDATA** command was specified.

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.

Notes

MERGE and **ALIAS** are mutually exclusive. Any attempt to use both commands in the same session will result in an error.

You cannot combine a common type section with other sections.

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 that 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 at the beginning of the combined section only.

Notes

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:

```
opt         case

sect        sect1,data
dcb.b       1,3

sect        sect2,data,align=4
dcb.b       1,4

sect        sect3,text,align=1
nop
nop
dcb.b       1,5

end
```

and **moduleb.src** has the following:

```
      opt         case

      sect        sect1,data
      dcb.b       1,3

      sect        sect2,data,align=1
      dcb.b       1,4

      sect        sect3,text,align=4
      nop
      nop
      scb.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 linked with the command line:

```
      lnk68k -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   4  (LONG)
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       sect1:00000200  modulea.o
                 sect2:00000008       sect2:00000008
                 sect3:00000000       sect3:00000002
moduleb          sect1:00000210       sect1:00000210  moduleb.o
                 sect2:00000010       sect2:00000010
                 sect3:00000004       sect3:00000006

START ADDRESS:    00000000

Link Completed
```

ALIGNMOD — Sets Alignment for Module Sections

Syntax

`ALIGNMOD {`*sect_name* `| (`*sect_type*`)}=`*align_value*

Description

sect_name	Specifies section name.
sect_type	A section type enclosed in parentheses. Any type associated with *sect_type* is aligned. *sect_type* can be one of the following:

	`CODE`
	`DATA`
	`ROMDATA`

align_value	Specifies a constant that 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 can be used to increase the alignment attribute of the module sections of the named module.

Notes

The alignment of a given combined section is the largest of its inclusive module sections.

If the alignment of a section is affected by both the **ALIGNMOD** *sect_name* and **ALIGNMOD** *sect_type* commands, then the **ALIGNMOD** *sect_name* alignment is used.

BASE — Specifies Location to Begin Loading

Syntax

`BASE` *number*

Description

number	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.

Note

Using commands such as **PUBLIC** and **BASE** without a **CHIP** command can cause errors. These commands are evaluated before any modules are loaded, and the default chip type (68000) and maximum address ($FFFFFF) will be applied to the addresses specified by these commands.

Examples

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

class	Specifies one of the following class names:

`PUBLICS`	All public and external names.
`MODULES`	All module names.
`SECTIONS`	All section 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 indicated in *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**).

If you do not specify a **CASE** or **NOCASE** command, the default will be **CASE**.

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, 68EC000, 68HC000, 68HC001, 68008, 68010, 68020, 68EC020, 68030, 68EC030, 68040, 68060, 68EC040, 68EC060, 68302, 68851, 68881, 68882, 68330, 68331, 68332, 68333, 68340, 68349, 68360, CPU32, and CPU32P. (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 5, *Relocation*, in this manual). All absolute addresses that appear in later commands or object modules are checked against the bounds established by the **CHIP** command.

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 that 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.

Note

Using commands such as **PUBLIC** and **BASE** without a **CHIP** command can cause errors. These commands are evaluated before any modules are loaded, and the default chip type (68000) and maximum address ($FFFFFF) will be applied to the addresses specified by these commands.

Table 11-3 summarizes the memory addressing among the 68000 family chips.

Table 11-3. 68000 Family Memory Addressing

Chip	Maximum Address	Bits	High Short Addressable Area of Memory
68000	$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	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68EC020	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68030	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68EC030	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68040	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68EC040	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68060	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68EC060	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68302	$FFFFFF	24	$FF8000 to $FFFFFF
68330	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68331	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68332	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68333	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68340	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68349	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68360	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
68851	not applicable		
68881	not applicable		
68882	not applicable		

(cont.)

Table 11-3. 68000 Family Memory Addressing (cont.)

Chip	Maximum Address	Bits	High Short Addressable Area of Memory
CPU32	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF
CPU32P	$FFFFFFFF	32	$FFFF8000 to $FFFFFFFF

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. Specifying an address bus width greater than that allowed by the processor is allowed. For example, **CHIP 68000, 31** is legal.

Examples

```
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.

```
* 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
S70500000000FA
```

```
* 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*
`[`*command*`];`*comment*

Description

comment	Indicates text that you want the linker to ignore.
command	Indicates any valid linker command.

An asterisk (*****) in column one denotes the beginning of a comment. A semicolon (**;**) in any column specifies the rest of the line as a comment.

Example

```
* Linker example.
*
*
LOAD mod1.o       ; This is the only module needed.
END
; Done
```

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
* noncommon 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 NOINTERNALS** has been specified, the **DEBUG_SYMBOLS** command will have no effect on the map file. Similarly, if **LISTABS NOINTERNALS** has been specified, the **DEBUG_SYMBOLS** command will have no effect on the absolute file. For **DEBUG_SYMBOLS** to take effect, **INTERNALS** must be turned on by either the **LISTABS** or **LISTMAP** command.

DEFINE — Defines a Symbolic Constant for Linker Commands

Syntax

`DEFINE` *symbol*`=`*const*`[,`*symbol*`=`*const*`]`. . .

Description

symbol	Defines a linker symbol.
const	Indicates a numerical constant or a previously defined symbol.

The **DEFINE** command assigns a constant value to a symbolic name. This symbol is only used by other linker commands and does not show up in the object symbol table or override any public defines.

Example

```
DEFINE      WORD=2
DEFINE      LONG=4
DEFINE      ONE_K=$1000
DEFINE      USER_DATA=$1000
DEFINE      SUPER_DATA=$4000

ALIGNMOD    data1=WORD
ALIGNMOD    data2=WORD
ALIGNMOD    code1=LONG
ALIGNMOD    code2=LONG

SECT        data1=USER_DATA
ORDER       data1,data2

SECT        data3=SUPER_DATA
ORDER       data3,data4

ORDER       code1,code2
ALIGN       code1=ONE_K
ALIGN       code2=ONE_K
```

In this example, `data1` will start at 1000 hexadecimal and will be followed by `data2`. The module sections within both `data1` and `data2` will be aligned on a 2-byte boundary. Section `data3` starts at 4000 hexadecimal followed by `data4`. The module section with both `data3` and `data4` will be aligned according to assembler specifications. The two code sections will start on 1000 hexadecimal boundaries following `data4`, and their module sections will be aligned on 4-byte boundaries.

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** command specifies that the message number indicated by *number* is to be treated as an error. Specifying **NOERROR** indicates that the message should not treated as an error.

This command has 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 **ERROR** or **NOERROR** command.

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.

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.

Example

```
EXTERN g1
LOAD module1.o,module2.o,textern.lib
END
```

In this example, the symbol `g1` is not defined in both `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

type Specifies the output format generated by the linker. Valid values are:

`HP`	Specifies HP 64000 object module format (HP-OMF).
`IEEE`	Specifies Microtec's 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` option.
`S[2|3]`	Specifies Motorola S-record file format. `S2` contains a 24-bit offset, while `S3` contains a 32-bit offset.
`INCREMENTAL`	Specifies incremental linking. Unless overridden with the *type* option, the output file format will be IEEE-695.

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.

Notes

Only one *type* option can be specified. In addition, `NOABS` cannot be used with the `INCREMENTAL` option. If either of these conditions 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:

```
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
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 ?A`*n*`,`*sectname*`[,`*offset*`]`

Description

`A`*n*	Specifies an address register **A**n, where *n* = 2, 3, 4, or 5.
sectname	Specifies a relocatable section.
offset	Specifies a number to be added to the load address of the relocatable section specified.

The **INDEX** command informs the linker of the run-time value of an address register that is normally used in relation to compiler code generation. The value you associate with a particular **A***n* 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 **?A***n*. 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 that combine relocatable expressions and address register indirection. For example, consider the following assembler syntax:

<relocatable_expression>`(A`*n*`)` or `(`*<relocatable_expression>*`,A`*n*`)`

Operands with the syntax shown 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:

<effective_address> = **A***n* + *disp*
displacement = *<effective_address>* - **A***n*
displacement = *<relocatable_expression>* - **A***n*

The **INDEX** command makes **A***n* 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 that 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.

Notes

See Chapter 4, *Instructions and Address Modes*, in this manual for more information on how the **INDEX** command can be used with array addressing for registers **A2** through **A5**.

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).
```

INITDATA — Creates ROM Section for RAM Initialization

Syntax

`INITDATA` {*sectname* | (*sect_type*)} [, {*sectname* | (*sect_type*)}]...

Description

sectname	Specifies a section name.
sect_type	A section type enclosed in parentheses. The type can be one of the following:

```
CODE
DATA
ROMDATA
```

The **INITDATA** command, in conjunction with the system routine **initcopy()**, allows data to be copied from ROM to RAM during initialization. This feature is intended for use on applications that have initialized values for data, which means that the data must 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 while the application is executing.

The **INITDATA** command automatically creates a new data section in ROM called **??INITDATA** and fills **??INITDATA** with a copy of all the initialized data values contained in the sections named as command arguments. If a section type is specified as a command argument, then all sections of the specified type are copied into the **??INITDATA** section.

When you use this command, you also should call a separate initialization routine, **initcopy()**, at the beginning of program execution. **initcopy()** invokes the function that copies initialized data from the **??INITDATA** section (in ROM) to the sections named as command arguments (in RAM). Thus, calling **initcopy()** at the start of the program reinitializes data variables each time your program runs. **initcopy()** 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** can be ordered and assigned an address using standard linker commands. You 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.** operator was used).

Notes

The **INITDATA** command is legal only during a final link.

Example

Assume you have the following memory map:

```
$1000-$2000       ROM: Program code
$2000-$4000       ROM: Initialized data for the section
                  "initvalues"
$8000-$A000       RAM: Run time location of the section
                  "initvalues"
```

You want to copy the initialization values for the section `initvalues` from ROM at locations `$2000-$4000` to RAM at locations `$8000-$A000` at startup. The following linker command file fragment will locate the section initvalues at `$8000` in ROM, and create a new section **??INITDATA** at location `$2000` in ROM, which contains the initialization values for the initvalues section. You must then make sure **initcopy()** is called from the **_START** function in `csys.c` (this is the default). **initcopy()** will copy the initialization vales from **??INITDATA** to initvalues.

```
SECT initvalues=$8000   ; address of section for data is in RAM
INITDATA initvalues     ; set up section for initialization
SECT ??INITDATA=$2000   ; put the initialization values here in
                        ; ROM
```

INTFILE — Stores Information Using Intermediate File or Virtual Memory

Syntax

```
[NO]INTFILE
```

Description

The linker, like the assembler, is a two-pass program. Intermediate information is stored using virtual memory (default for non-PC hosts) between pass 1 and 2. The **INTFILE** command lets you store this intermediate information in a temporary file. The **NOINTFILE** command lets you store this information using virtual memory. The default for PC hosts is **INTFILE**, and the default for all other hosts is **NOINTFILE**.

The **INTFILE** command has a global effect. Multiple **INTFILE** or **NOINTFILE** commands are not allowed in a command file. If multiple **INTFILE** or **NOINTFILE** commands are encountered, an error or warning will be issued, and the first **INTFILE** or **NOINTFILE** command that was encountered will be in effect.

Notes

With different systems, using a temporary file may be faster than using virtual memory. Also, depending on the configuration for running large jobs, the virtual allocation size can be limited. If a virtual error is returned, an error message will be displayed. You can try to run the program using the **INTFILE** command, which then produces an intermediate file as opposed to using virtual memory.

Example

```
INTFILE
LOAD mod1.o
LOAD mod2.o
END
```

LISTABS — Lists Symbols to Output Object Module

Syntax

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

Description

option	Specifies 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. (default: **INTERNALS** for IEEE; default: **NOINTERNALS** for S-record)
`[NO]PUBLICS`	Places globally defined symbols into the output object module. (default: **NOPUBLICS** for S-record, **PUBLICS** for IEEE.)

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. This command has no effect when used with HP-OMF files.

The **FORMAT** command will affect the default settings of the **INTERNALS** and **PUBLICS** options. The defaults for the format types are shown in Table 11-4.

Table 11-4. LISTABS Default Settings

Format Type	Symbol Types	
IEEE	internals	publics
S	nointernals	nopublics

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.

The effect of the **LISTABS NOPUBLICS** command during an incremental link is to fully resolve any public symbols possible. Once resolved, these symbols are unavailable to the symbol resolution process in subsequent link steps.

LISTMAP — Specifies Layout and Content of the Map

Syntax

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

Description

option Specifies 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: **NOINTERNALS**)
`[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_ADDR` or `/ADDR` is specified, the public symbol table is listed in address order. If `/BY_NAME` or `/NAME` is specified, the public symbol table is listed in alphabetical order. This command must be typed without spaces between the **INTERNAL** and **PUBLICS** keywords and the **/** qualifiers. (default: **NOPUBLICS**)
`LENGTH` *lval*	Specifies that the map file page length is set to *lval* lines where *lval* is a numeric value between 5 and 255. (default: **55** lines including any header information generated by the linker)
`[NO]RETAINED [/DELETED]`	Causes a listing of public and

local mergeable regions of code or data by symbol name. These regions are used for Microtec compiler support. The default is to list those instances that were kept, along with their addresses and which modules they came from. If `/DELETED` is specified, deleted regions are shown along with the modules that they belonged to.
(default: **RETAINED**)

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.

LISTMAP RETAINED will cause the linker to execute slowly. This option may be turned off if it is not necessary to know the addresses of retained mergeable regions.

LOAD — Loads Specified Object Modules

Syntax

`LOAD [-]` *module* `[,[-]` *module*`]...`

Description

module	Specifies a file that contains the object module. Input object modules can consist of relocatable modules from the assembly process, relocatable modules from incremental linking, or libraries. A library preceded by a minus sign (**-**) will cause all modules in the library to be loaded. A minus sign preceding a non-library module has no effect. Special characters in *module* should be escaped.

The **LOAD** command specifies one or more input object modules to be loaded or searched 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 10, *Linker Operation*, in this manual 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 (search will stop if absolute path is specified and the file is not found)
2. Path specified in the **MRI_68K_LIB** environment variable
3. Path specified in the **USR_MRI** environment variable, with a **/lib** (UNIX) or **\lib** (DOS) appended

4. **/usr/mri/lib** (UNIX hosts only)

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 the specified modules and retains the public symbol definitions and debugging information for the files. The code and data information is omitted from the output object module. Except for omitting code, data, and external references, **LOAD_SYMBOLS** performs identically to the **LOAD** command.

The files named in *module* will be searched in the following order:

1. Absolute path (search will stop if an absolute path is specified and the file is not found)
2. Relative path
3. Relative to the **MRI_68K_LIB** environment variable

See the *User's Guide* for more information on the library path environment variable.

If a *module* is a library, **LOAD_SYMBOLS** retains all external symbols in each loaded module. All forward references within the library cause allocation of space for subsequent modules. If *module* is not a library, all external symbols will be ignored.

Examples

This example shows the **lnkcmd1.cmd** linker command file:

```
* The three modules each contain three sections:
* SECT1, SECT2, and SECT3.
*
LOAD MAIN.OBJ,COMM.OBJ
LOAD_SYMBOLS overlay_root, romlib
*
```

This example causes all of the information regarding `MAIN` and `COMM` to be loaded, but only the section space for `SECT1`, `SECT2`, `SECT3` and any public definitions and debugging information in `overlay_root` and `romlib` are to be loaded.

This example shows two linker command files: **lnkcmd2.cmd** and **lnkcmd3.cmd**:

```
format s
listabs nointernals nopublics
absolute rom_code
sect rom_code=$8000
load rom_module.o
end

format s
listabs nointernals nopublics
sect code=$1000
sect rom_code=$8000
load_symbols rom_module.o
load application.o
end
```

These two linker command files show how code, which has been compiled in separate modules, can be linked at specific addresses and placed into ROM. These specific modules must be identified before compilation and should be isolated in separate C source files. At compile time, the code section for these files is renamed using a compiler command line option. For this example, the code that will be downloaded into ROM is placed into a code section named `rom_code`. For UNIX or DOS hosts, the compiler command line would be:

```
mcc68k -c -NTrom_code rom_module.c
```

The `-c` compiler option is needed to prevent the linker from being automatically invoked.

When the linker is separately invoked, the **lnkcmd2.cmd** linker command file creates an output file for the code destined for ROM (previously identified as `rom_code`). The output file contains Motorola S-records (`format s`) with no symbol information (`listabs nointernals nopublics`). The `absolute` command tells the linker to place only the contents of the `rom_code` section in the output file.

The second linker command file, **lnkcmd3.cmd**, deals with the code that will not be placed in ROM. The `application.o` code module refers to symbols in `rom_module.o`. With the `load_symbols` command, the linker treats `rom_module.o` as if it were a normal part of the link, adjusting all relocatable addresses and resolving external references to the module. However, it does not place any code for this module in the output file. This command file produces an absolute output file that contains only the code from the module `application.o` with resolved references to the section `rom_code`.

If changes are made later to any code in `application.o`, the new code can be relinked using the **lnkcmd3.cmd** linker command file to produce a new absolute file for downloading into the target system without having to alter the permanent ROM.

These command files can be altered to create absolute modules for debugging using the XRAY Debugger by altering the `format` command (to create modules readable by the XRAY Debugger) and the `listabs` command (to include global and local symbols):

```
format ieee
listabs publics, internals
```

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` *new_sect merge_arg*`[,`*merge_arg*`]...`

Description

new_sect	Specifies the name of the new, combined section.
merge_arg	Specifies one of the following:

sectname	A section name.
{*sectname*, *module*}	A section name followed by a module name. The braces are required.

Both *sectname* and *module* can be replaced by the wildcard 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 *new_sect* already exists and it is not specified in the *merge_arg*() sections, the sections specified in *merge_arg*(s) will be placed before *new_sect*, and the final combined sections will use the name *new_sect*. Multiple **MERGE** commands with the same *new_sect* are concatenated.

Notes

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 the section *Command Position Dependencies* in this chapter for a description of the order in which commands are executed relative to other commands.

MERGE cannot be used with common sections.

Example

```
* There are three modules each containing three
* sections: SECT1, SECT2, SECT3.
*
MERGE SECTA SECT1,{SECT2,MOD1},{SECT3,MOD3}
MERGE SECTA {SECT3,MOD2}
*
SECT SECTA=$1000
SECT SECT2=$2000
SECT SECT3=$3000
*
LOAD MOD1,MOD2,MOD3
*
* This causes a new section with the name SECTA to
* be created. It is located at $1000 and contains
* 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. Any symbol assigned a value by the **PUBLIC** command during the link is placed in this module, which is defined during the link.

If this command is not used to specify a name, the name of the output module will be specified by the rules outlined in Chapter 2, *UNIX/DOS User's Guide*.

The user-specified name is included in the S-records only when the **LISTABS PUBLICS** command is in the command file.

Notes

This command has no effect on the name of the output object file.

Example

```
NAME READER
FORMAT IEEE
LISTABS PUBLIC
LOAD mod1.obj,mod2.obj
END
```

ORDER — Specifies Section Order

Syntax

ORDER {*sectname* | (*sect_type*)} [, {*sectname* | (*sect_type*)}] ...

Description

sectname	Specifies a section name.
sect_type	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 10, *Linker Operation*, in this manual, the normal order of the sections is the order in which the linker encountered the section 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 loca-

tion of these sections will also set the starting point at which the linker will begin looking for the starting point for the next section named in the command.

5. If the order that 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 missing, a warning is issued, and the linker will continue with the next named section, if any.

Notes

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 plus sign (+). A continuation character must be placed between section names and will not continue a name to the next line.

Examples

```
ORDER SEC1,COMSEG
ORDER SECT1 #
  SECT2,SECT3 #
  SECT4

ORDER (CODE), (DATA), zerovars

ORDER sect1       ;these ORDER commands are
ORDER sect3       ;identical to ORDER sect1,sect3,sect5
ORDER sect5

ORDER SEC1,COMSEG
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.

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 that 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.

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     ALIGN

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

symbol1	Specifies a name for the public symbol being defined.
value	Specifies a constant number *value* to be assigned to the public symbol *symbol1*. *value* is treated as absolute and will not be relocated relative to any section base.
symbol2	Specifies another symbol whose value, relocation type, and section ID are assigned to *symbol1*.
offset	Specifies a constant number that is added or subtracted from *symbol2*.

The **PUBLIC** command defines and/or changes the value of an external definition. The external symbols are defined at load time, which may prevent reassembly. This command is position-independent.

Note

Using commands such as **PUBLIC** and **BASE** without a **CHIP** command can cause errors. These commands are evaluated before any modules are loaded, and the default chip type (68000) and maximum address ($FFFFFF) will be applied to the addresses specified by these commands.

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.

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.

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 will not appear in 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

```
RESMEM $200,$100
RESADD $2,$101
LOAD module1
END
```

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 will not appear in 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.

Example

The command file:

```
RESMEM $200,$100
RESADD $2,$101
LOAD module1
END
```

will reserve two regions in memory.

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.

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

sname	Specifies a section name.
size	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*.

Notes

It is an error to define a size 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**.

It is an error to define a section size less than the default size.

Example

```
SECTSIZE stack=$100
```

SORDER — Specifies Short Section Order

Syntax

`SORDER [`*sectname* `| (`*sect_type*`)] [, {`*sectname* `| (`*sect_type*`)} ]` ...

Description

sectname	Specifies a section name.
sect_type	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., the order in which the linker encountered their names).

The **SORDER** command 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 into 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 the next section named in the command.

5. If the order which the **SORDER** command specified cannot be maintained, then 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 missing, a warning is issued, 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. It is not assigned either the common or the noncommon attribute, so it can be either attribute.

Notes

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 the long attribute to newly found sections.

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 in 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

ORDER   SECT1 #
  SECT2,SECT3 #
  SECT4
```

START — Specifies Output Module Starting Address

Syntax

`START` *value*

Description

value	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. If no starting address is specified, the starting address is obtained from the input modules created using the **END** assembler directive in the **END** record of the main program. 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.

Example

```
START $7FC
```

SYMTRAN — Transforms Public/External Symbols

Syntax

```
SYMTRAN /EMBEDDED "string1" ["string2"]
SYMTRAN [/LEADING] "string1" ["string2"]
SYMTRAN /TRUNCATE num
```

Description

`/EMBEDDED`	Indicates that each occurrence of the character pattern "*string1*" (in the order it appears in the symbol) in each public symbol will have that pattern changed to "*string2*". If "*string2*" is absent, all occurrences of the pattern in the symbol are simply deleted. This is useful when combining code written with different leading character conventions.
string1	Specifies the character pattern to transform.
string2	Specifies the replacement character pattern.
`/LEADING`	Indicates that each public symbol whose leading characters matches the pattern in "*string1*" will have that pattern changed to "*string2*". If "*string2*" 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`	Truncates each public symbol whose length exceeds *num*.
num	Truncates symbols to the number of characters specified by *num*.

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 "ABA" "X"
PUBLIC ABABABABABA ; ABABABABABA -> XBABABABABA

SYMTRAN /TRUNCATE 4
PUBLIC _SYM_1_ ; _SYM_1_ -> _SYM
```

These examples show the usage of the `SYMTRAN` command on several hypothetical `PUBLIC` symbols.

UPPERCASE — Shifts Names to Uppercase

Syntax

`UPPERCASE [`*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 **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.

```
   . . .
   . . .
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.

Notes

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 320
```

Downgrades error condition `320` to a warning.

Sample Linker Session 12

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 two.

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 12-1 shows the linker command file, and Figure 12-2 shows the output listing. Figure 12-3 shows the linker absolute file, while Figures 12-4 to 12-6 show the assembly 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 12-1 shows the command file that 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 place 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 12-2 shows the link map with 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 that 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:

   a. High-level elements are compiler symbols whose attribute is `LOCAL`. The `OFFS/ADDR` column indicates the stack address offset in bytes for each section. High-level symbols contain both `MODULE` and `FUNCTION` information.

   b. Low-level elements are assembler symbols whose attribute is `ASMVAR`. `OFFS/ADDR` is the actual section address. Only the `MODULE` information is listed in the local symbol table.

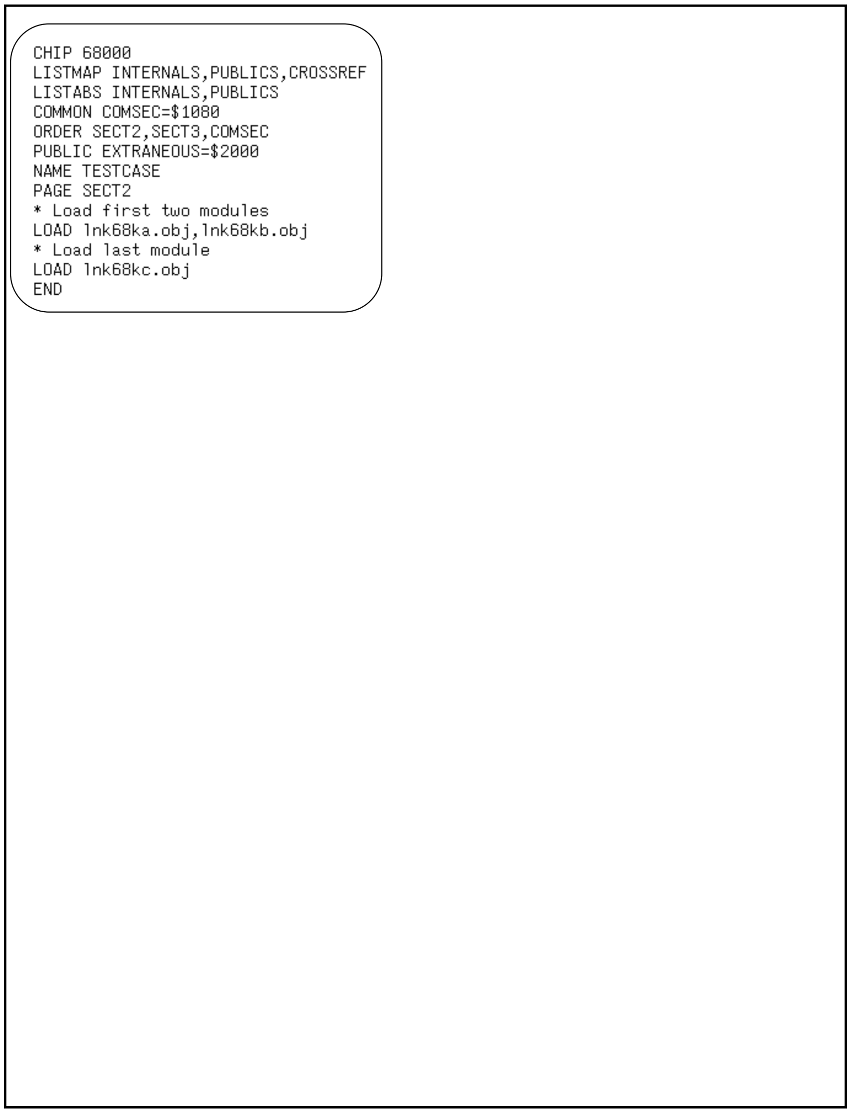
6. The public symbol table contains the list of **PUBLIC** symbols, the section, the actual section address, and the modules. When a module name cannot be specified or does not exist, a **$$** symbol will be used.

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.

Linker Command File



1 ⟶

```
CHIP 68000
LISTMAP INTERNALS,PUBLICS,CROSSREF
LISTABS INTERNALS,PUBLICS
COMMON COMSEC=$1080
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 12-1. Linker Command File

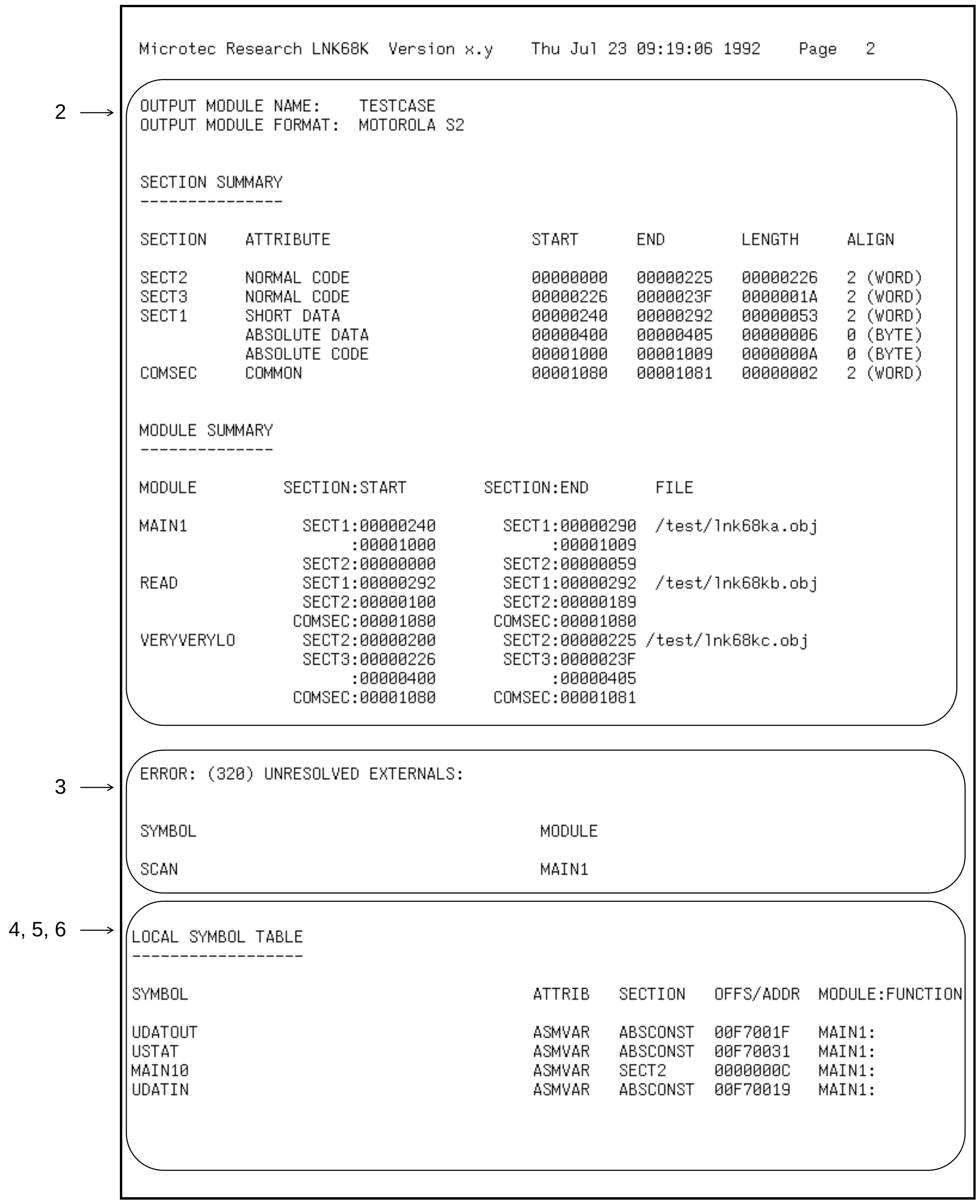
Linker Map File

```
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 →

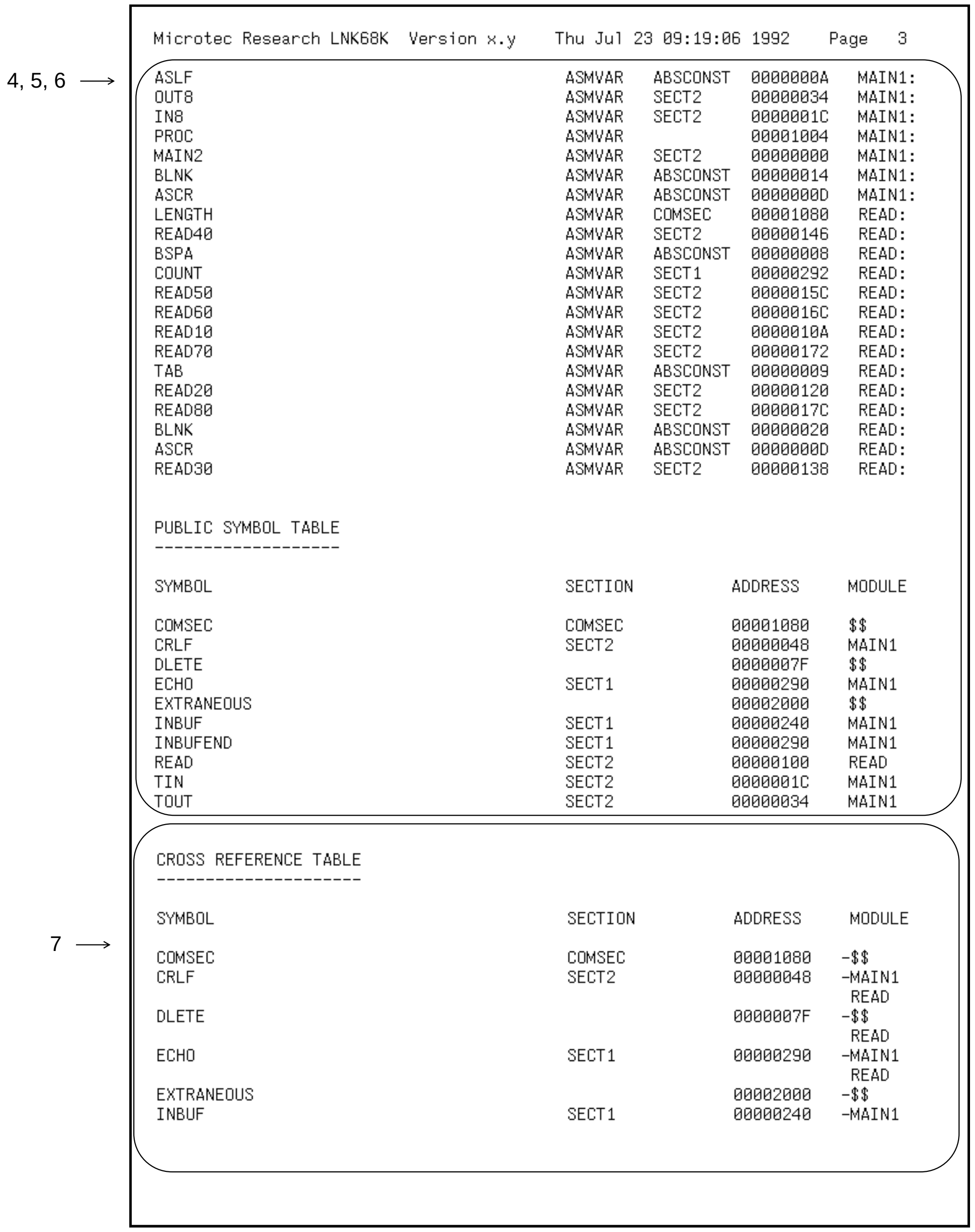
```
CHIP 68000
LISTMAP INTERNALS,PUBLICS,CROSSREF
LISTABS INTERNALS,PUBLICS
COMMON COMSEC=$1080
ORDER SECT2,SECT3,COMSEC
PUBLIC EXTRANEOUS=$2000
NAME TESTCASE
PAGE SECT2
* Load first two modules
LOAD lnk68ka.obj,lnk68kb.obj
* Load last module
LOAD lnk68kc.obj
END
```

Figure 12-2. Linker Map File



```
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   0000000A   0 (BYTE)
COMSEC     COMMON                        00001080   00001081   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             SECT1:00000292       SECT1:00000292  /test/lnk68kb.obj
                 SECT2:00000100       SECT2:00000189
                COMSEC:00001080      COMSEC:00001080
VERYVERYLO       SECT2:00000200       SECT2:00000225 /test/lnk68kc.obj
                 SECT3:00000226       SECT3:0000023F
                      :00000400            :00000405
                COMSEC:00001080      COMSEC:00001081

ERROR: (320) UNRESOLVED EXTERNALS:


SYMBOL                                    MODULE

SCAN                                      MAIN1

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 12-2. Linker Map File (cont.)



```
Microtec Research LNK68K  Version x.y    Thu Jul 23 09:19:06 1992    Page   3

ASLF                                      ASMVAR   ABSCONST  0000000A   MAIN1:
OUT8                                      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    00001080   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     0000010A   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           00001080    $$
CRLF                                      SECT2            00000048    MAIN1
DLETE                                                      0000007F    $$
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
---------------------

SYMBOL                                    SECTION          ADDRESS     MODULE

COMSEC                                    COMSEC           00001080   -$$
CRLF                                      SECT2            00000048   -MAIN1
                                                                       READ
DLETE                                                      0000007F   -$$
                                                                       READ
ECHO                                      SECT1            00000290   -MAIN1
                                                                       READ
EXTRANEOUS                                                 00002000   -$$
INBUF                                     SECT1            00000240   -MAIN1
```

Figure 12-2. Linker Map File (cont.)

```
Microtec Research LNK68K  Version x.y   Thu Jul 23 09:19:06 1992    Page   4

                                                                     READ
INBUFEND                                  SECT1            00000290   -MAIN1
READ                                      SECT2            00000100   -READ
                                                                     MAIN1
SCAN                                                       00000000   -$$
                                                                     MAIN1
TIN                                       SECT2            0000001C   -MAIN1
                                                                     READ
TOUT                                      SECT2            00000034   -MAIN1
                                                                     READ


START ADDRESS:   00000000

Load Completed


 Errors: 1, Warnings: 0
```

8, 9 ⟶

Figure 12-2. Linker Map File (cont.)

Linker Absolute File

The absolute output module is shown in Figure 12-3. In this sample, the output object file shown is in Motorola S-record absolute hexadecimal format

```
$$ TESTCASE
  COMSEC $00001080 EXTRANEOUS $00002000
$$ MAIN1
  TIN $0000001C INBUF $00000240 INBUFEND $00000290 ECHO $00000290
  CRLF $00000048 DLETE $0000007F TOUT $00000034 UDATOUT $00F7001F
  USTAT $00F70031 MAIN10 $0000000C UDATIN $00F70019 ASLF $0000000A
  OUT8 $00000034 IN8 $0000001C PROC $00001004 MAIN2 $00000000
  BLNK $00000014 ASCR $0000000D
$$ READ
  READ $00000100 LENGTH $00001080 READ40 $00000146 BSPA $00000008
  COUNT $00000292 READ50 $0000015C READ60 $0000016C READ10 $0000010A
  READ70 $00000172 TAB $00000009 READ20 $00000120 READ80 $0000017C
  BLNK $00000020 ASCR $0000000D READ30 $00000138
$$ VERYVERYLONGMODULENAME
$$
S00600004844521B
S20E00100060000022E7C00001000C5
S2140000004EB900000100227C0000024012190C01CB
S214000010001467F84EB9000000004E75323900F73C
S2140000200031080100016 6F4143900F700190202D5
S214000030007F4E75323900F700310801000266F481
S21400004013C200F7001F4E75143C000D4EBAFFE6B3
S20E000050143C000A4EBAFFDE4E759F
S214000100227C0000024042780292 4EB90000001C99
S2140001100C4100186600000A4EB90000004860E076
S2140001200C41000D66000012343802926 7DC228112
S21400013033C2000010804E750C01007F6600001E62
S2140001403438029267C4538904380001029 2343C62
S21400015000084EB90000003460000018 0C010009C9
S2140001606700000A0C0100206D00000812C15278DA
S214000170029234380292 0C42005067CA323802901B
S20E00018067884EB900000034608066
S2140002004E71323900001080D27804004E7133C12E
S21400021000001081 6A0000060A0100AA12380403D2
S2140002204E714E4F66FC227C000004003219 6600B8
S21400023000084EB90000021C0000021C000004006A
S20A0004000506021C4E7109
S5030015E7
S804000000FB
```

Figure 12-3. Linker Absolute File

Assembly Listings for the Linker Example

Figures 12-4 to 12-6 show three assembly listings of programs that are combined by the linker. The actual load is shown in Figure 12-4. Note the following points when examining the assembly listings:

1. The `MAIN` program in Assembly Listing 1 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. Assembly Listing 2 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. Assembly Listing 3 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

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                                      LLEN     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                               ;End of buffer
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                   ;Read next line
31   00000006 227C 0000 0000  R         MOVE.L  #INBUF,A1              ;Start of buffer
32   0000000C 1219              MAIN10  MOVE.B  (A1)+,D1
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                   ;Get value
36   0000001A 4E75                      RTS
37                              *
38                              * NAME   - IN8
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 CHARCTER
49                              *
50                              *
```

Figure 12-4. Assembly Listing 1

```
Microtec Research ASM68K   Version x.y     Wed Jul 22 17:08:16 1992    Page  2

Line Address
0000001C 3239 00F7 0031    IN8     MOVE.W  USTAT,D1              ;Read UART status
00000022 0801 0001                 BTST.L  #$1,D1                ;Check if ready
00000026 66F4                      BNE     IN8                   ;Not ready yet
00000028 1439 00F7 0019            MOVE.B  UDATIN,D2             ;Read data
0000002E 0202 007F                 ANDI.B  #DLETE,D2             ;Delete parity bit
00000032 4E75                      RTS
                           *
                           * NAME - OUT8
                           *
                           * THIS ROUTINE WRITES A CHARACTER TO THE TERMINAL
                           *
                           * ENTRY PARAMETERS
                           *
                           * D2  - CHARACTER TO OUTPUT
                           *
                           * EXIT PARAMETERS
                           *
                           *  NONE
                           *
00000034 3239 00F7 0031    OUT8    MOVE.W  USTAT,D1              ;Read status
0000003A 0801 0002                 BTST.L  #2,D1                 ;Check if ready
0000003E 66F4                      BNE     OUT8                  ;Not ready
00000040 13C2 00F7 001F            MOVE.B  D2,UDATOUT            ;Output data
00000046 4E75                      RTS
                           *
                           * NAME - CRLF
                           *
                           * THIS ROUTINE OUTPUTS A CARRIAGE RETURN AND LINE FEED
                           *
                           *
00000048 143C 000D         CRLF    MOVE.B  #ASCR,D2
0000004C 4EBA FFE6                 JSR     OUT8
00000050 143C 000A                 MOVE.B  #ASLF,D2
00000054 4EBA FFDE                 JSR     OUT8
00000058 4E75                      RTS
                           *
                           *
00F70031                   USTAT   EQU     $F70031             ;UART status
00F7001F                   UDATOUT EQU     $F7001F             ;Serial output port
00F70019                   UDATIN  EQU     $F70019             ;Serial input port
0000000D                   ASCR    EQU     13
0000000A                   ASLF    EQU     10
00000014                   BLNK    EQU     20
0000007F                   DLETE   EQU     $7f
0000001C                 R TIN     EQU     IN8
00000034                 R TOUT    EQU     OUT8
                                   END        MAIN2
```

Figure 12-4. Assembly Listing 1 (cont.)

```
Microtec Research ASM68K   Version x.y    Wed Jul 22 17:08:16 1992    Page  3


                   Symbol Table

Label             Value

ASCR            0000000D
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 12-4. 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: /usr4/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, 0= NO ECHO
22                              *
23                              * EXIT PARAMETERS
24                              *
25                              *  INBUF - CONTAINS INPUT LINE
26                              *
27                              *
28                                      LLEN      80
29                              READ        IDNT
30                                      OPT       D,E,T
31                                      XDEF      READ
32                                      XREF      CRLF,TIN,TOUT
33                                      XREF.S    INBUF,INBUFEND,ECHO,DLETE
34                              *
35                              *
36                                      SECT.S    SECT1
37   00000000                   COUNT   DS.B      1
38                              *
39                              *
40                                      SECT      SECT2
41                              *
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 6600 000A                 BNE       READ20
47   00000018 4EB9 0000 0000  E         JSR       CRLF
48   0000001E 60E0                      BRA       READ
49   00000020 0C41 000D         READ20  CMP       #ASCR,D1
50   00000024 6600 0012                 BNE       READ30
```

Figure 12-5. 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     #DLETE,D1
57   0000003C 6600 001E                 BNE       READ50
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 0008                 MOVE      #BSPA,D2
63   00000052 4EB9 0000 0000  E         JSR       TOUT
64   00000058 6000 0018                 BRA       READ70
65   0000005C 0C01 0009         READ50  CMP.B     #TAB,D1
66   00000060 6700 000A                 BEQ       READ60
67   00000064 0C01 0020                 CMP.B     #BLNK,D1
68   00000068 6D00 0008                 BLT       READ70
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 READ80  MOVE.W    ECHO,D1
75   00000080 6788                      BEQ       READ10
76   00000082 4EB9 0000 0000  E         JSR       TOUT
77   00000088 6080                      BRA       READ10
78                              * PLACE VARIABLES IN COMMON
79                                      COMMON    COMSEC
80   00000000                   LENGTH  DS.B      1
81   0000000D                   ASCR    EQU       13
82   00000008                   BSPA    EQU       8
83   00000020                   BLNK    EQU       $20
84   00000009                   TAB     EQU       $09
85                                      END
```

Figure 12-5. Assembly Listing 2 (cont.)

```
Microtec Research ASM68K   Version x.y    Thu Jul 23 09:49:04 1992    Page  3


                   Symbol Table

Label              Value

ASCR             0000000D
BLNK             00000020
BSPA             00000008
COUNT     SECT1 :00000000
CRLF             External
DLETE            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 12-5. Assembly Listing 2 (cont.)

Assembly Listing 3

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                                * NAME - VERYVERYLONGMODULENAME
3                                *
4                                * THIS ROUTINE DOES NOT HAVE ANY EXTERNAL
5                                * REFERENCES TO THE OTHER MODULES, BUT
6                                * CREATES SEVERAL RELOCATABLE SEGMENTS
7                                * AND IS LINKED WITH THEM FOR DEMONSTRATION
8                                * PURPOSES.
9                                *
10                               *
11                               VERYVERYLONGMODULENAME  IDNT
12                                       OPT     E
13                               *
14                               * Create a Relocatable Section
15                               *
16                                       SECT    SECT2
17   00000000 4E71                       NOP
18   00000002 3239 0000 0000  R          MOVE.W  LEN1,D1
19   00000008 D278 0400 4E71             ADD     DATA,D1
20   0000000E 33C1 0000 0001  R          MOVE    D1,LEN2
21   00000014 6A00 0006                  BPL     LAB1
22   00000018 0A01 00AA                  EORI.B  #$AA,D1
23   0000001C 1238 0403 4E71     LAB1    MOVE.B  DATA+3,D1
24   00000022 4E4F               LAB3    TRAP    #15
25   00000024 66FC                       BNE     LAB3
26                               *
27                               * Create a Relocatable Section
28                               *
29                                       SECT    SECT3
30   00000000 227C 0000 0400             MOVE.L  #DATA,A1
31   00000006 3219                       MOVE    (A1)+,D1
32   00000008 6600 0008                  BNE     SKP
33   0000000C 4EB9 0000 001C  R          JSR     LAB1
34   00000012 0000 001C       R SKP      DC.L    LAB1
35   00000016 0000 0400                  DC.L    DATA
36                               *
37                               * Create an Absolute Section
38                               *
39                                       ORG     $400
40   00000400 0506               DATA    DC.B    5,6
41   00000402 001C            R          DC      LAB1
42   00000404 4E71                       NOP
43                               *
44                               * Place variables in COMMON
45                               *
46                                       COMMON  COMSEC
47   00000000                    LEN1    DS.B    1
48   00000001                    LEN2    DS.B    1
49                                       END
```

Figure 12-6. Assembly Listing 3

```
Microtec Research ASM68K   Version x.y     Thu Jul 23 09:37:30 1992    Page  2


                   Symbol Table

Label            Value

DATA           00000400
LAB1    SECT2 :0000001C
LAB3    SECT2 :00000022
LEN1    COMSEC:00000000
LEN2    COMSEC:00000001
SKP     SECT3 :00000012
```

Figure 12-6. Assembly Listing 3 (cont.)

Librarian Operation 13

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.

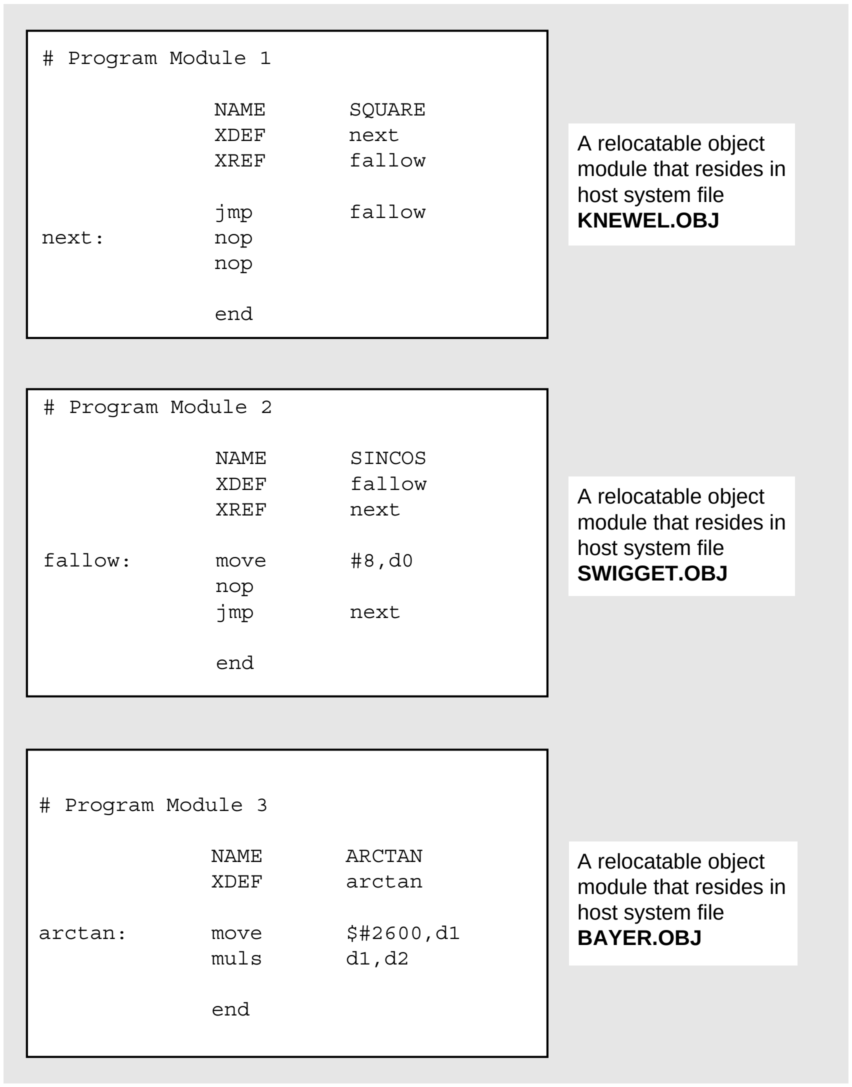
When used in connection with the librarian, the word ‘‘module’’ refers to a relocatable object module that results from assembling a source program with the ASM68K Assembler.

Error messages and warnings are listed in Appendix D, *Librarian Error Messages*, in this manual.

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 13-1 shows three relocatable object modules that result from an assembly.



```
# Program Module 1

             NAME       SQUARE
             XDEF       next
             XREF       fallow

             jmp        fallow
next:        nop
             nop

             end
```

A relocatable object module that resides in host system file **KNEWEL.OBJ**

```
# Program Module 2

             NAME       SINCOS
             XDEF       fallow
             XREF       next

fallow:      move       #8,d0
             nop
             jmp        next

             end
```

A relocatable object module that resides in host system file **SWIGGET.OBJ**

```
# Program Module 3

             NAME       ARCTAN
             XDEF       arctan

arctan:      move       $#2600,d1
             muls       d1,d2

             end
```

A relocatable object module that resides in host system file **BAYER.OBJ**

Figure 13-1. 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.

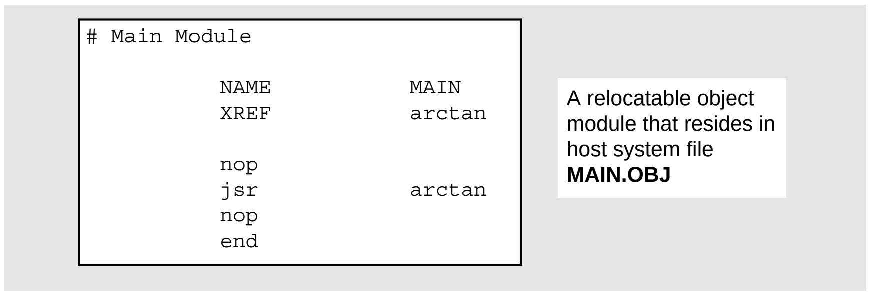
The relocatable modules illustrated consist of load data information, relocation information, and records that indicate public symbols and external symbols.

By using various combinations of librarian commands, the relocatable object modules shown can be made members of a library. For example, a new library can be created by using the following commands (Chapter 2, *UNIX/DOS User's Guide*, contains a 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 13-2 is in a host system file named **MAIN.OBJ**. This module has a reference to the public symbol `arctan`.



```
# Main Module

          NAME          MAIN
          XREF          arctan

          nop
          jsr           arctan
          nop
          end
```

A relocatable object module that resides in host system file **MAIN.OBJ**

Figure 13-2. 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 13-3).

```
# Main Module

      NAME          MAIN
      XREF          arctan
      XREF          fallow

      nop
      jsr           arctan
      jsr           fallow
      end
```

A relocatable object module that resides in host system file **MAIN.OBJ**

Figure 13-3. 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 **JMP** 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*, in this manual.

Librarian Commands 14

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 a **CLEAR** 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 maximum symbol length is 512 characters; however, use of symbols longer than 127 characters is not recommended.

The librarian recognizes the special characters listed in Table 14-1.

Table 14-1. 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 only members of a library.
- The plus sign (**+**) followed by a carriage return acts as a continuation character and allows 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:

The command:

```
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. Include comments by using a 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. Table 14-2 gives an alphabetical listing of the librarian commands.

Table 14-2. 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	Defines new library
DE	DELETE	Deletes module(s) from current library
DI	DIRECTORY	Lists library modules
EN/ Q	END/ QUIT	Terminates librarian execution
EXT	EXTRACT	Copies library module to a file
F	FULLDIR	Displays library or the contents of the library module
H	HELP	Displays command syntax and can be used in a context-sensitive manner
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_name*`[(`*module_name*`[,`*module_name*`]...)]`

Description

library_name	Specifies the library where the named modules reside.
module_name	Indicates a relocatable object module to be added to the library named in a previous **OPEN** or **CREATE** command.

The **ADDLIB** command adds one or more object modules from one library to the library currently being created or modified.

You can include the entire library by entering the *library_name* and no module names.

Notes

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 this example, `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 this example, 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 this example, `CLEAR` must be executed before opening and processing `LIBRARY3.LIB`.

CREATE — Defines New Library

Abbreviation

CR

Syntax

`CREATE` *library_name*

Description

library_name	Specifies the name of the library file being created.

The **CREATE** command defines 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.

Notes

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 this example, 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 you use 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_name*`]`

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 names, directory information is displayed for the named modules only.
list_name	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.

Notes

Object module names are case-sensitive.

Related Commands

FULLDIR

Example

```
DIRECTORY sieve.lib
```

In this example, all modules in `sieve.lib` and their sizes are listed:

```
Library sieve.lib
Module           Size  Processor
SIEVE .....      386   68000 family
MODULE1 ...      397   68000 family
MODULE ....      289   68000 family

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.

Notes

If you do not issue a **SAVE** command, the librarian will not build a new library.

Related Commands

SAVE

Example

```
DIR  NEW.LIB
END
```

In this example, 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 three file names created for this example are `MODA.OBJ`, `MODB.OBJ`, and `MODC.OBJ`. File name extensions listed in *File Name Defaults* in Chapter 2, *UNIX/DOS 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

library_name	Specifies the library file whose contents are to be listed. If you enter just the *library_name*, the contents of all modules are listed.
module_name	Specifies the module whose contents are to be listed. When you enter specific *module_names*, information is displayed for the named modules only.
list_filename	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

In this example, 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.

Notes

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 this example, the library named `MATH.LIB` is opened.

REPLACE — Replaces Library Module

Abbreviation

R

Syntax

`REPLACE` *module_name*`[,`*module_name*`]` . . .

Description

module_name	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.

Notes

REPLACE mod1 is not the same as **DELETE mod1** followed by **ADDMOD mod1**. **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 this example, 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, `NEW.LIB` is a newly created library comprised of the relocatable object modules named `REL1.OBJ`, `REL2.OBJ`, and `FORTUN.OBJ`.

Sample Librarian Session 15

Overview

The sample librarian test programs and output listings in this chapter describe the input command file and the output listing format.

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

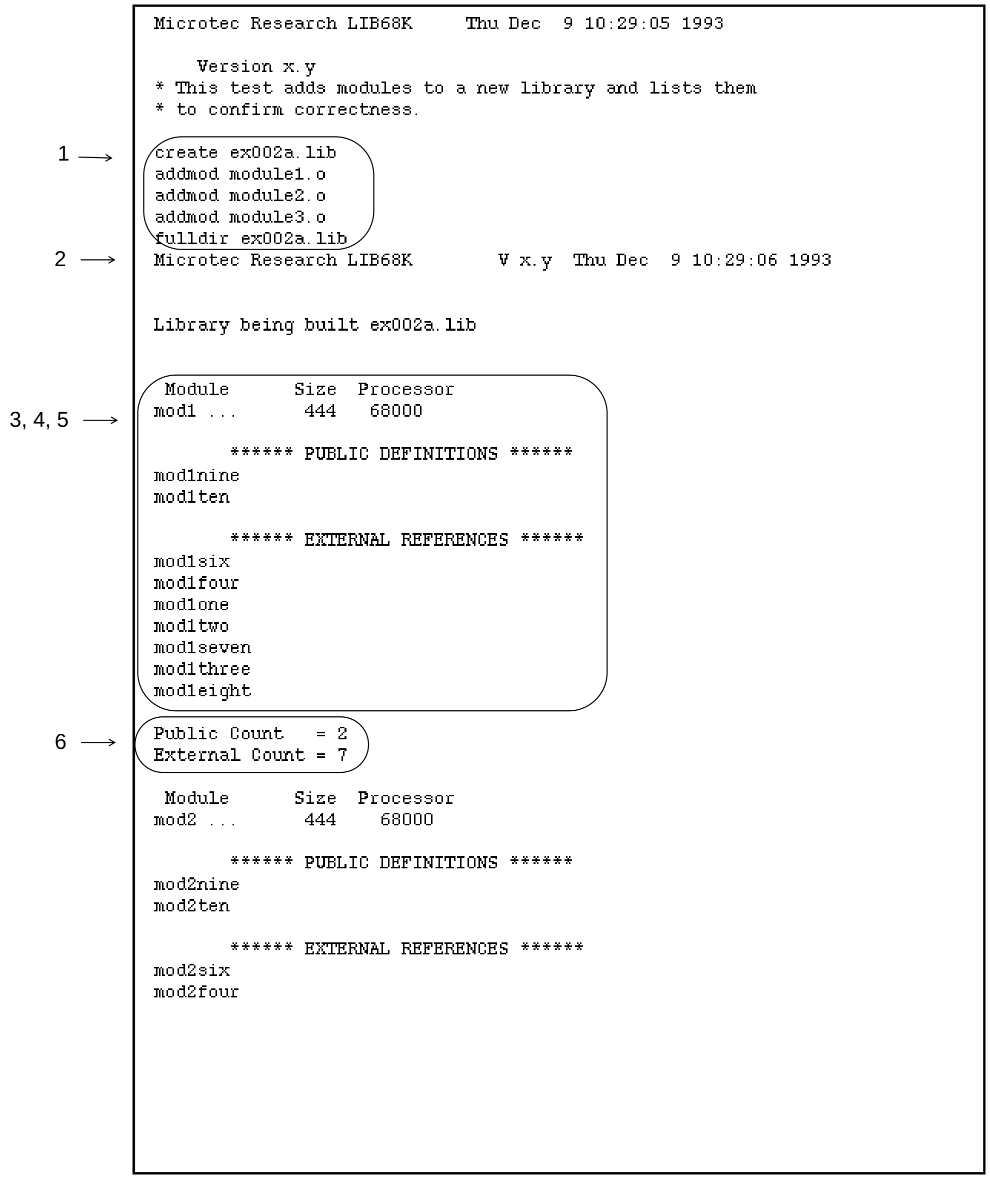
Sample Program 1 is shown in Figure 15-1.

Sample Program 1 Description

The output listing contains the following information.

1. The command file is listed.
2. A new library, `ex002a.lib`, is created based on the **CREATE** command in the command file.
3. Three modules, `module1.o`, `module2.o`, and `module3.o`, 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.

Sample Program 1 Output Listing



```
Microtec Research LIB68K     Thu Dec  9 10:29:05 1993

    Version x.y
* This test adds modules to a new library and lists them
* to confirm correctness.

create ex002a.lib
addmod module1.o
addmod module2.o
addmod module3.o
fulldir ex002a.lib
Microtec Research LIB68K        V x.y  Thu Dec  9 10:29:06 1993


Library being built ex002a.lib


 Module      Size  Processor
mod1 ...      444   68000

       ****** PUBLIC DEFINITIONS ******
mod1nine
mod1ten

       ****** EXTERNAL REFERENCES ******
mod1six
mod1four
mod1one
mod1two
mod1seven
mod1three
mod1eight
Public Count   = 2
External Count = 7

 Module      Size  Processor
mod2 ...      444    68000

       ****** PUBLIC DEFINITIONS ******
mod2nine
mod2ten

       ****** EXTERNAL REFERENCES ******
mod2six
mod2four
```

Figure 15-1. Librarian Sample Program 1 Output Listing

```
mod2one
mod2two
mod2seven
mod2three
mod2eight

Public Count   = 2
External Count = 7

 Module      Size  Processor
mod3 ...      444    68000

       ****** PUBLIC DEFINITIONS ******
mod3nine
mod3ten

       ****** EXTERNAL REFERENCES ******
mod3six
mod3four
mod3one
mod3two
mod3seven
mod3three
mod3eight

Public Count   = 2
External Count = 7
Module Count   = 3
end
```

7 ⟶

Figure 15-1. Librarian Sample Program 1 Output Listing (cont.)

Librarian Sample Program 2

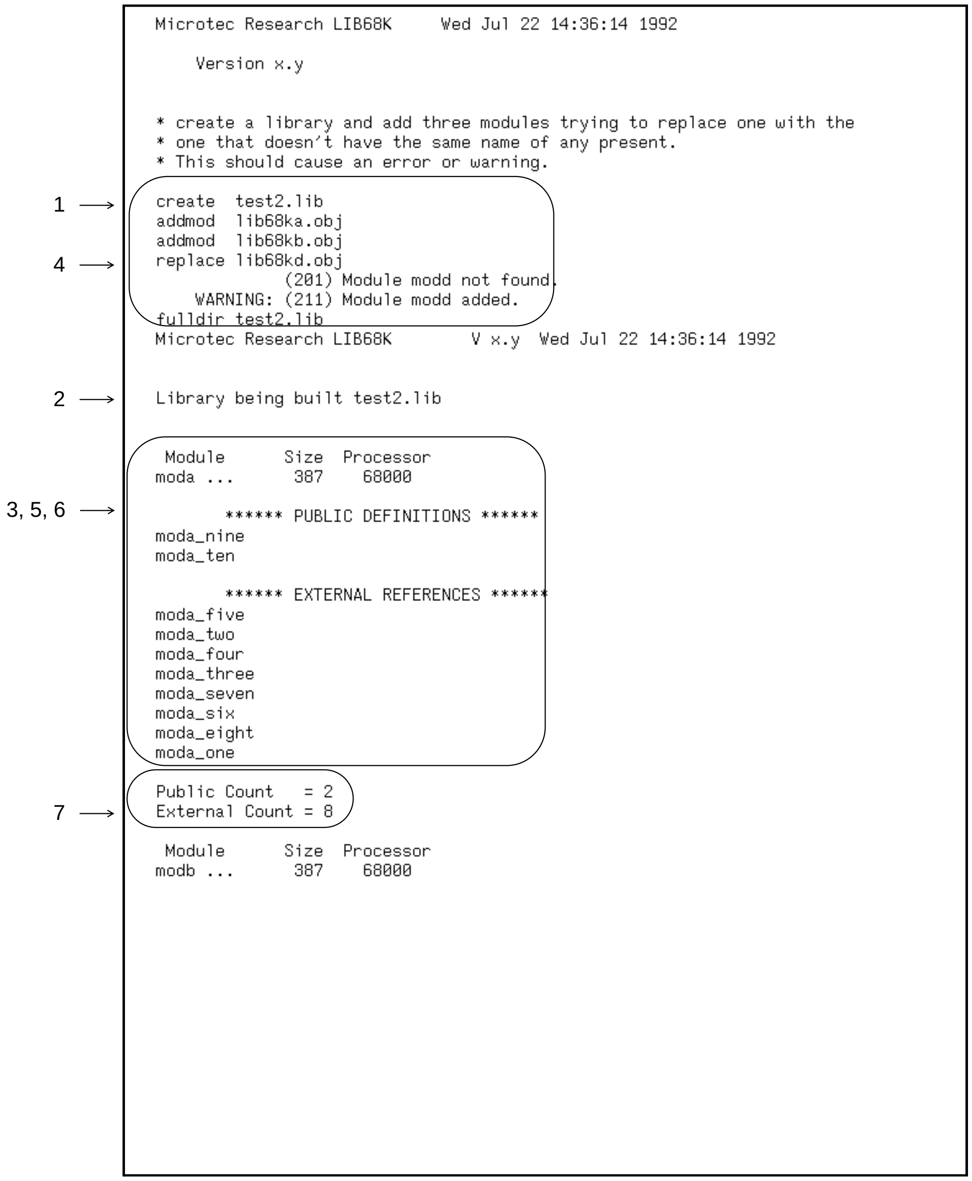
Sample Program 2 is shown in Figure 15-2.

Sample Program 2 Description

Refer to the following points in the text and the sample listing:

1. The command file is listed with error or warning messages following any commands that could not be executed.
2. A new library, `test2.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.

Sample Program 2 Output Listing



```
Microtec Research LIB68K     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.
create  test2.lib
addmod  lib68ka.obj
addmod  lib68kb.obj
replace lib68kd.obj
             (201) Module modd not found.
    WARNING: (211) Module modd added.
fulldir test2.lib
Microtec Research LIB68K        V x.y  Wed Jul 22 14:36:14 1992


Library being built test2.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

Public Count   = 2
External Count = 8

 Module      Size  Processor
modb ...      387    68000
```

Figure 15-2. 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 15-2. Librarian Sample Program 2 Output Listing (cont.)

ASCII Character Set: Appendix A

Table A-1. ASCII Character Set

Name	Octal	Numer-icDeci-mal	Hex	Description
NUL	00	0	0	NULL
SOH	01	1	1	Start of heading
STX	02	2	2	Start of text
ETX	03	3	3	End of text
EOT	04	4	4	End of transmission
ENQ	05	5	5	Inquiry
ACK	06	6	6	Acknowledge
BEL	07	7	7	Bell
BS	010	8	8	Backspace
HT	011	9	9	Horizontal tabulation
LF	012	10	A	Line feed
VT	013	11	B	Vertical tabulation
FF	014	12	C	Form feed
CR	015	13	D	Carriage return
SO	016	14	E	Shift out
SI	017	15	F	Shift in
DLE	020	16	10	Data link escape
DC1	021	17	11	Device control 1
DC2	022	18	12	Device control 2

(cont.)

Table A-1. ASCII Character Set (cont.)

Name	Octal	Numer-icDeci-mal	Hex	Description
DC3	023	19	13	Device control 3
DC4	024	20	14	Device control 4
NAK	025	21	15	Negative acknowledge
SYN	026	22	16	Synchronous idle
ETB	027	23	17	End of block
CAN	030	24	18	Cancel
EM	031	25	19	End of medium
SUB	032	26	1A	Substitute
ESC	033	27	1B	Escape
FS	034	28	1C	File separator
GS	035	29	1D	Group separator
RS	036	30	1E	Record separator
US	037	31	1F	Unit separator
SP	040	32	20	Space
!	041	33	21	Exclamation point
"	042	34	22	Double quote
#	043	35	23	Pound sign
$	044	36	24	Dollar sign
%	045	37	25	Percent sign
&	046	38	26	Ampersand
’	047	39	27	Apostrophe (single quote)
(	050	40	28	Left parenthesis
)	051	41	29	Right parenthesis

(cont.)

Table A-1. ASCII Character Set (cont.)

Name	Octal	Numer-icDeci-mal	Hex	Description
*	052	42	2A	Asterisk
+	053	43	2B	Plus
,	054	44	2C	Comma
-	055	45	2D	Hyphen (minus)
.	056	46	2E	Period (decimal point) / dot
/	057	47	2F	Slant (slash)
0	060	48	30	Zero
1	061	49	31	One
2	062	50	32	Two
3	063	51	33	Three
4	064	52	34	Four
5	065	53	35	Five
6	066	54	36	Six
7	067	55	37	Seven
8	070	56	38	Eight
9	071	57	39	Nine
:	072	58	3A	Colon
;	073	59	3B	Semicolon
<	074	60	3C	Less than / Left angle bracket
=	075	61	3D	Equals sign
>	076	62	3E	Greater than / Right angle bracket
?	077	63	3F	Question mark
@	0100	64	40	"At" sign

(cont.)

Table A-1. ASCII Character Set (cont.)

Name	Octal	Numer-icDeci-mal	Hex	Description
A	0101	65	41	Upper-case A
B	0102	66	42	Upper-case B
C	0103	67	43	Upper-case C
D	0104	68	44	Upper-case D
E	0105	69	45	Upper-case E
F	0106	70	46	Upper-case F
G	0107	71	47	Upper-case G
H	0110	72	48	Upper-case H
I	0111	73	49	Upper-case I
J	0112	74	4A	Upper-case J
K	0113	75	4B	Upper-case K
L	0114	76	4C	Upper-case L
M	0115	77	4D	Upper-case M
N	0116	78	4E	Upper-case N
O	0117	79	4F	Upper-case O
P	0120	80	50	Upper-case P
Q	0121	81	51	Upper-case Q
R	0122	82	52	Upper-case R
S	0123	83	53	Upper-case S
T	0124	84	54	Upper-case T
U	0125	85	55	Upper-case U
V	0126	86	56	Upper-case V
W	0127	87	57	Upper-case W

(cont.)

Table A-1. ASCII Character Set (cont.)

Name	Octal	Numer-icDeci-mal	Hex	Description
X	0130	88	58	Upper-case X
Y	0131	89	59	Upper-case Y
Z	0132	90	5A	Upper-case Z
[	0133	91	5B	Left square bracket
\	0134	92	5C	Backslash
]	0135	93	5D	Right square bracket
^	0136	94	5E	Caret
_	0137	95	5F	Underscore
`	0140	96	60	Backquote
a	0141	97	61	Lower-case a
b	0142	98	62	Lower-case b
c	0143	99	63	Lower-case c
d	0144	100	64	Lower-case d
e	0145	101	65	Lower-case e
f	0146	102	66	Lower-case f
g	0147	103	67	Lower-case g
h	0150	104	68	Lower-case h
i	0151	105	69	Lower-case i
j	0152	106	6A	Lower-case j
k	0153	107	6B	Lower-case k
l	0154	108	6C	Lower-case l
m	0155	109	6D	Lower-case m
n	0156	110	6E	Lower-case n

(cont.)

Table A-1. ASCII Character Set (cont.)

Name	Octal	Numer-icDeci-mal	Hex	Description
o	0157	111	6F	Lower-case o
p	0160	112	70	Lower-case p
q	0161	113	71	Lower-case q
r	0162	114	72	Lower-case r
s	0163	115	73	Lower-case s
t	0164	116	74	Lower-case t
u	0165	117	75	Lower-case u
v	0166	118	76	Lower-case v
w	0167	119	77	Lower-case w
x	0170	120	78	Lower-case x
y	0171	121	79	Lower-case y
z	0172	122	7A	Lower-case z
{	0173	123	7B	Left curly brace
|	0174	124	7C	Vertical bar
}	0175	125	7D	Right curly brace
~	0176	126	7E	Tilde
DEL	0177	127	7F	Delete

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.

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.

Note

All error numbers for the ASM68K Assembler are prefixed with the letter **A**. For example, the “no error” message is numbered as **A500**. The following list will contain only the numbers that will follow the initial **A**.

Assembler Messages and Errors

500 No error

501 Missing argument

The assembler expected an argument to be provided for the statement. Check the syntax of the statement.

502 Operator expected but not found

The assembler expected an operator but none was provided. Check the syntax of the statement. See Table 3-6 in Chapter 3, *Assembly Language*, in this manual for a summary of operators the assembler accepts.

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. See Chapter 6, *Assembler Directives*, in this manual for the correct symbol or character.

504 Right parenthesis not valid in this context

A right parenthesis was included in a statement but not expected. Check the syntax of the statement. See Chapter 4, *Instructions and Address Modes*, in this manual for examples of the correct use of parentheses.

505 Operator not valid in this context

An operator was used in a statement where it is not valid. Check the syntax of the statement. See Chapter 3, *Assembly Language*, in this manual for more information on operators.

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

The assembler expected an operand for an operator but none was provided. Check the syntax of the statement. See Chapter 3, *Assembly Language*, in this manual for more information on operators.

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. This error will also occur if there are spaces in an expression.

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 Bad or missing operand

The expression is probably too complicated. You should make the expression shorter by substituting a series of simpler expressions for the longer, more complex one.

511 Invalid operands for \ operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that 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 that 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 that 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 that 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 that 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 that 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 that 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 that 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 that 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 that 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 that 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 that 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 that 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 that is evaluated to an absolute value by the assembler.

525 Invalid character

An unexpected character or a character that is not in the set supported by the assembler was found. See Chapter 3, *Assembly Language*, in this manual to determine if the character is supported.

526 Closing string delimiter missing

A string must be enclosed in either single (**'***string***'**) or double quotation marks (**"***string***"**). An opening quotation mark was entered without a closing quotation mark.

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

An opcode that the assembler does not recognize was entered. Check the opcode against the list provided in Chapter 4, *Instructions and Address Modes*, in this manual.

529 Invalid opcode/qualifier combination

530 Undefined symbol

A symbolic name in the operand field that was found was never defined. The symbol should have been previously defined for certain directives but was not, although the symbol might have been defined after the directive. If the symbol has not been defined anywhere in the program, fix the error by defining the symbol.

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 that was outside the range for that particular instruction or directive. Check the *Reference Manual* or the manufacturer's manual to determine the legal range.

538 Label required on this directive

A directive was found that required a label preceding the directive. Refer to *Assembler Statements* in Chapter 3, *Assembly Language*, in this manual to determine what constitutes a legal label for this particular directive.

539 Duplicate IDENT directive (ignored)

540 Relocatable expression invalid in this context

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.

541 Comma expected but not found

The assembler expected a comma in a statement, but none was provided. Check the syntax of the statement that caused the error.

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. See Chapter 3, *Assembly Language*, in this 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

A symbol that contains an illegal character was entered. See Chapter 3, *Assembly Language*, in this manual for a description of the types of symbols allowed.

547 This sort of symbol cannot be made an external definition

Either the symbol has already been declared a section, a module name, an external, or a register, or the symbol has been defined by the **XDEF** directive. Fix the error by not allowing the symbol to have an external declaration.

548 Invalid external symbol

The symbol has already been declared a section, a module name, public, or a register. Fix the error by not allowing the symbol to have an external declaration.

549 Value will be sign-extended to 32 bits at runtime

550 Unable to open Include file

The assembler could not locate the file to be included. Make sure you have entered the file name correctly in the **INCLUDE** directive or on the command line.

551 Invalid formal parameter name

The rules surrounding the definition of formal parameters for a macro have been violated. See Chapter 7, *Macros*, in this 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: label not permitted on this directive.

The assembler found a label that was not permitted for this particular directive. Check the *Reference Manual* for the correct syntax regarding 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 directive option

You have specified an option for the directive that the assembler cannot recognize. See Chapter 6, *Assembler Directives*, in this 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.

561 A colon-separated pair of registers is invalid in this context

Register pairs cannot be separated by a colon in this instruction.

562 Register expected but not found

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 (e.g., **D5-D0**). 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

A general purpose register was encountered where a special purpose register was required (or vice versa), or an illegal expression was encountered following the register expression.

567 Left parenthesis expected but not found

The assembler expected a left parenthesis. Make sure that every left parenthesis has a corresponding right parenthesis.

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

A label without a colon was found. This message is a warning. Terminate all label fields with a colon.

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 is 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 that has too many operands. See Chapter 4, *Instructions and Address Modes*, in this manual to determine the number of operands allowed for the particular instruction.

588 This combination of operands is not valid for this instruction

An invalid addressing mode has been found in one of the operands. See Chapter 4, *Instructions and Address Modes*, in this manual to determine the addressing mode allowed for the particular mnemonic.

589 Too few bytes allocated on Pass 1 for forward reference

The assembler has found a situation where the forward reference range is too large. Verify that none of the forward references exceed the accessible address ranges of their respective instructions.

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

This error usually indicates an error condition that you have defined. Normally this directive is used to indicate an invalid macro call. You should determine why the assembler assembled this source line.

592 Register list required for REG directive operand

593 This directive invalid outside a macro

The **ENDM** directive was encountered outside of a macro. As these directives have no meaning outside of a macro, you should determine why these statements were assembled. There may be a missing **MACRO** directive.

594 This character invalid within real constant

An invalid character was found while the assembler was reading a floating-point constant. See Chapter 3, *Assembly Language*, in this 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/Include files nested too deeply

The assembler has found a situation in which macros or include files are nested too deeply. This normally indicates an endless recursive loop. Verify that recursive macro or include calls are capable of terminating.

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) or **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

These extensions are 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 that is position-dependent in the source file.

639 Use of MERGE_START/MERGE_END directive in non-relocatable section

The **MERGE_START** and **MERGE_END** directives may only be used in relocatable sections. Use of these in a section designated with the **ORG** directive is not valid.

640 MERGE_START directive missing

A **MERGE_END** directive must always follow a **MERGE_START** directive. Nesting of these directives is not allowed.

641 MERGE_END directive missing

A **MERGE_END** directive must always follow a **MERGE_START** directive. Nesting of these directives is not allowed.

642 Label associated with MERGE_START directive not within bounds

The symbol used with the **MERGE_START** directive must exist between the **MERGE_START** directive and its matching **MERGE_END** directive.

643 Label associated with MERGE_START directive not at start of boundary

Normally, the symbol used with the **MERGE_START** directive is the first directive that follows the **MERGE_START** directive itself. This is normally an entry point into a function or the address of a variable.

644 Displacement size may be too small for reference to mergeable text

This warning indicates that a reference was made to a symbol that is between **MERGE_START** and **MERGE_END** directives. The offset used with this instruction is smaller than the maximum offset size. As a result, the linker might not be able to store the correct offset due to not having enough bytes in the instruction. If possible, increase the size of the instruction offset.

645 Empty merge directive ignored

No instructions or data were found between a pair of **MERGE_START** and **MERGE_END** directives. Therefore, these directives were ignored.

646 The -h and -H command line options (for HP 64000 support) will no longer be available in the next release

This is only a warning.

647 Incompatible usage: label not permitted on this directive

A label was used with a directive that does not accept labels. The label will be ignored.

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.

Note

All error numbers for the LNK68K Linker are prefixed with the letter **A**. For example, the “bad IEEE object record” message is numbered as **A300**. The following list will contain only the numbers that will follow the initial **A**.

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. Verify that the assembler made the appropriate assumptions.
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 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. This definition of the public symbol will be ignored.

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 *Reference Manual*.

313 Invalid Operand

An operand specified for a command contains invalid characters, does not exist, or is too large.

Note

If this error occurred with the **PUBLIC** command, try specifying a **CHIP** command before the **PUBLIC** command.

314 Chip inconsistency

The linker has encountered a file assembled with a **CHIP** directive that 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 that 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 what your values are being set to.

All values are evaluated as unsigned 32-bit values. These values are expected to be within 8 bits sign-extended (i.e., $FFFFFF80 to $FFFFFFFF or 0 to $7F) displacements that will be sign-extended to 32 bits at run time (e.g., the operand of **MOVEQ**). In the more common case of immediate values that are not sign-extended at run time, the expected range is 0 to $FF or $FFFFFF00 to $FFFFFFFF.

In either case, the value inserted in the object module is the low 8 bits of the complete 32-bit value, whether this error is reported or not. This message interrupts the 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 what your values are being set to.

For example, an absolute short instruction refers to a location that is not in the range $0 through $7FFF or $FFFF8000 through $FFFFFFFF. A PC-plus-16-bit displacement instruction may refer to a location that is more than +/- 32K bytes from the present location.

Often this error occurs in conjunction with an **Unresolved External** error. The 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 run time, then the value must fall within the range $0 through $7FFF or

$FFFF8000 through $FFFFFFFF. If the field will not be sign-extended, the value must fall in the range $0 through $FFFF or $FFFF0000 through $FFFFFFFF.

In any case, the value inserted into the field is the low 16 bits of the value.

324 Section Mismatch Between Symbol Def and Ref for Symbol

An **XREF** from the assembler had a section associated with it that 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 specified. For more information, refer to the installation instructions.

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 **INITDATA** 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 that 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
68020/30/40/60	$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’s latest version of the linker. Relink using the latest version of Microtec’s linker.

345 Duplicate Public (From Library)

The public symbol that caused this message to appear was already defined. Loading will continue, and the symbol will be listed on the link map. This definition of the public symbol will be ignored.

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 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 nonexistent 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 what your values are being set to.

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 $FFFFFFFF or 0 to $7FFFFF). At run time, these values will be sign-extended to 32 bits. In the more common case of immediate values that are not sign-extended at run time, 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 11, *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.

385 Illegal processor type

An unknown processor type was encountered while processing an IEEE-695 file. This object file is either corrupted, not a valid 68000 family object file, or not a Microtec IEEE-695 object file.

397 Bracket symbol cannot be an external reference

A compiler-generated symbol was used as an external reference, which is not allowed. These symbols can only be defined as external definitions.

398 No matching bracket symbol

A compiler-generated symbol occurred without a matching symbol. This should never occur in a normal program.

399 Mixed user/compiler template definitions

A C++ template was defined as both a generic instance and as a user-defined instance, but in different modules. Instances of a given template must be either all generic or all user-defined. It is not valid to mix the two together. The user-defined instance will be retained in this case.

400 Out of memory while optimizing; Optimizations suppressed

The linker was not able to allocate memory while processing C++ object files. Optimizations for removal of identical code could not be completed. More memory is needed.

401 Reserved memory overlap

Two reserved memory areas overlap each other. The commands should be modified to remove the overlap.

402 Empty File

An input file is empty, so the linker is unable to process it.

403 The HP 64000 Support will no longer be available in the next release

This is only a warning.

409 Erroneous IEEE type record syntax error

Conflicts have occurred with the type information (**-Zt**) and debug information compression (**-Zp**) options. Use the **-Znpt** switch to disable these options.

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.

Note

All error numbers for the LIB68K Librarian are prefixed with the letter **A**. For example, the “unable to open file *filename*” message is numbered as **A101**. The following list will contain only the numbers that will follow the initial **A**.

Message Severity Levels

There are two classes of errors that can occur during librarian 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 of one of the types listed in Table D-1.

Table D-1. Librarian Message Severity Levels

Type	Severity
Warning	Not fatal.
Message	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 Messages and Errors

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. The file name specified is either incorrect or the 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 included.

This message has a companion message that specifically states why the library file was not included.

120 Use HELP for proper command syntax.

This message suggests using the **HELP** librarian command, which will display the 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.

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 that occurs earlier in the session 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 that 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.

IEE2AOUT Error Messages: Appendix E

Introduction

This appendix describes the error messages and warnings that appear if errors are detected while running IEE2AOUT.

IEE2AOUT Messages and Errors

IEE2AOUT produces errors of the format:

`Error: (`*XXXX*`)` *string of message*

The number *XXXX* is the seek address (file offset) of the offending IEEE record in a bad IEEE file.

Note

The number *XXXX* preceding the string of the error message is not an error number.

There are three types of error messages produced by IEE2AOUT:

`Error: (`*XXXX*`) Internal ERROR. . .`

This message occurs when there is a conflict between **-T**, **-D**, **-B**, and **-I** options.

`Error: (`*XXXX*`)` *message arguments*

This message is formatted by the main program for IEE2AOUT. *message* is a description of the error situation. *arguments* is an optional string of numbers or other values to substantiate the message.

`Error: (`*XXXX*`) Record: (`*nn*`)` *XXXX RECORD*`:` *message arguments* *nn* is an error number between 1 and 18 or a warning number between 1 and 3. *RECORD* is a hex dump of the IEEE record. *message* is a description of the error situation. *arguments* is an optional string of numbers or other values to substantiate the message.

Glossary: Appendix F

Absolute Section	Part of an assembly program that is to be loaded at fixed locations in memory. It contains no relocatable information.
Address Expression	An expression whose value represents a location in memory.
Base Address	The lowest address considered for loading relocatable sections of the absolute object module.
Common Section	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.
Load Address	Memory address where the lowest byte of a section is placed.
Module	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.
Noncommon Section	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.
Numeric Expression	An expression whose value represents a number.
Register Expression	An address expression whose base or index attribute is not null.
Relocatable Section	A general purpose section that can contain both instructions and/or data.
Short Section	A section type. This type of section can be referenced by Absolute Short Addressing Mode.

Starting Address	Location where execution begins.
Subsection	Individual pieces of code from various modules that make up a section.

Object Module Formats: Appendix G

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 whose name comes from the **NAME** command for the linker.

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 (S0)

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 (S1)

A data record specifies data bytes that are to be loaded into memory. Figure G-1 shows the format for such a record. The columns shown in the figure represent half of a byte (4 bits).

```
1 2    3 4     5 6 7 8      9 ... 40     41  42

S ID   byte     load       data...data  checksum
       count   address      1      n
```

Figure G-1. Data Record Format for 16-Bit Load Address

Column	**Description**
1	Contains the ASCII character **S**, which indicates the start of a record in Motorola S-record format.
2	Contains the ASCII character identifying the record type. For data records, this character is 1.
3 to 4	Contain 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.
5 to 8	Contain 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, and columns 7 and 8 contain the low-order address byte.
9 to 40	Contain the specifications for up to 16 bytes of data.
41 to 42	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.

Data Record (S2)

A data record specifies data bytes that are to be loaded into memory. Figure G-2 shows the format for such a record. The columns shown in the figure represent half of a byte (4 bits).

1 2	3 4	5 6 7 8 9 10	11 ... 41 42	43 44
S ID	byte count	load address	data...data 1 *n*	checksum

Figure G-2. Data Record Format for 24-Bit Load Address

Column	Description
1	Contains the ASCII character **S**, which indicates the start of a record in Motorola S-record format.
2	Contains the ASCII character identifying the record type. For data records, this character is 2.
3 to 4	Contain 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.
5 to 10	Contain 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, and columns 9 and 10 contain the low-order address byte.
11 to 42	Contain the specifications for up to 16 bytes of data.
43 to 44	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.

Data Record (S3)

A data record specifies data bytes that are to be loaded into memory. Figure G-3 shows the format for such a record. The columns shown in the figure represent half of a byte (4 bits).

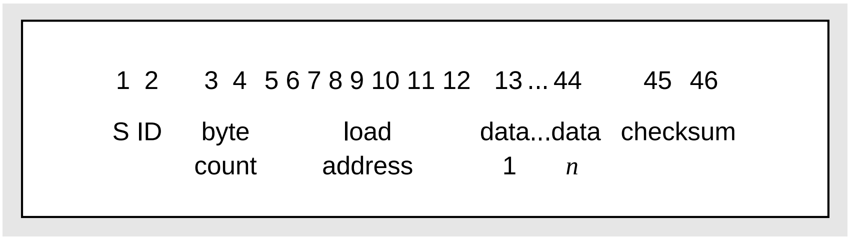


Figure G-3. Data Record Format for 32-Bit Load Address

Column	Description
1	Contains the ASCII character **S**, which indicates the start of a record in Motorola S-record format.
2	Contains the ASCII character identifying the record type. For data records, this digit is 3 for 32-bit addresses.
3 to 4	Contain 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.
5 to 12	Contain 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, and columns 11 and 12 contain the low-order address byte.
13 to 44	Contain the specifications for up to 15 bytes of data.
45 to 46	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 (S5)

The record count record verifies the number of data records preceding it. Figure G-4 shows the format for such a record. The columns shown in the figure represent half of a byte (4 bits).

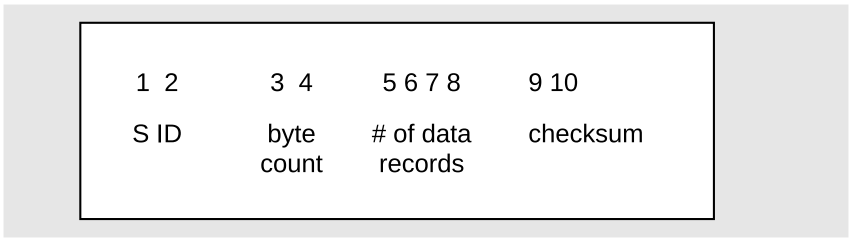


Figure G-4. Record Count Record Format

Column	Description
1	Contains the ASCII character **S**, which indicates the start of a record in Motorola S-record format.
2	Contains the ASCII character **5**, which indicates a record count record.
3 to 4	Contain the byte count, ASCII string **03**.
5 to 8	Contain the number of data records in this file. The high-order byte is in columns 5 and 6.
9 to 10	Contain 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`.

Terminator Record (S7)

A terminator record specifies the end of the data records. Figure G-5 shows the format for such a record. The columns shown in the figure represent half of a byte (4 bits).

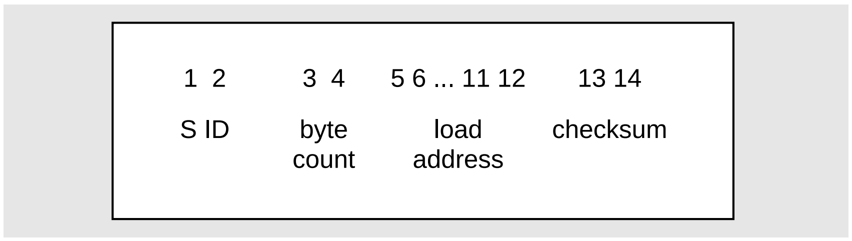


Figure G-5. Terminator Record Format for 32-Bit Load Address

Column	Description
1	Contains the ASCII character **S**, which indicates the start of a record in Motorola S-record format.
2	Contains the ASCII character **7**, which indicates a 32-bit load address.
3 to 4	Contain the byte count, ASCII string **04**.
5 to 12	Contain the load address that is either set to zero or to the starting address specified in the **END** directive or **START** command (there are no data bytes).
13 to 14	Contain the checksum for the record.

Terminator Record (S8)

A terminator record specifies the end of the data records. Figure G-6 shows the format for such a record. The columns shown in the figure represent half of a byte (4 bits).

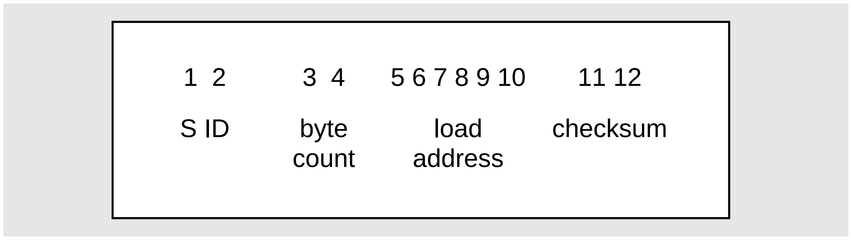


Figure G-6. Terminator Record Format for 24-Bit Load Address

Column	Description
1	Contains the ASCII character **S**, which indicates the start of a record in Motorola S-record format.
2	Contains the ASCII character **8**, which indicates a 24-bit load address.
3 to 4	Contain the byte count, ASCII string **04**.
5 to 10	Contain the load address, which is either set to zero or to the starting address specified in the **END** directive or **START** command. There are no data bytes.
11 to 12	Contain the checksum for the record.

Example:

```
S804000AF0001
```

The previous example shows a terminator record with a 24-bit load address of `0x000AF0` and a checksum of `0x01`.

Terminator Record (S9)

A terminator record specifies the end of the data records. Figure G-7 shows the format for such a record. The columns shown in the figure represent half of a byte (4 bits).

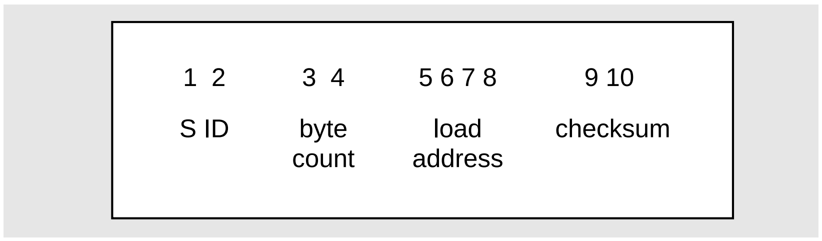


Figure G-7. Terminator Record Format for 16-Bit Load Address

Column	Description
1	Contains the ASCII character **S**, which indicates the start of a record in Motorola S-record format.
2	Contains the ASCII character **9**, which indicates a 16-bit load address.

3 to 4	Contain the byte count, ASCII string **04**.
5 to 8	Contain the load address, which is either set to zero or to the starting address specified in the **END** directive or **START** command (there are no data bytes).
9 to 10	Contain the checksum for the record.

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:

```
$$ testcase
  stack_top $00002000 extraneous $00003000
$$ main
 start $0000023E end_loop1 $0000023A end_loop2 $0000022D sieve_size
$00000064
 loop1 $0000020E inc_count $00000236 loop2 $0000021E main $00000200
  sieve $0000026E
$$ saveprime
  saveprime $00000246 prime_tab $000002D2 prime_count $000002F0
  max_prime $0000001E table_full $0000025E
$$ veryverylongmodulename
  len $000002F2 data $000002F6 len1 $000002F4 rel_lab $00000266
  lab1 $00001000
$$
S00600004844521B
S21400026E010101010101010101010101010101016B
S21400027E010101010101010101010101010101015B
S21400028E010101010101010101010101010101014B
S21400029E010101010101010101010101010101013B
S2140002AE010101010101010101010101010101012B
S2140002BE010101010101010101010101010101011B
S2080002CE0101010123
S21400020058000118024649000137002EA808480089
S2140002106437002 6A882F0026E0400370018A82271
S2140002204A00643C0007F2026E060020F1180246FF
S21400023049000137000 4A80820D4180000195F20E0
S20900024000180200 1981
S2140002D2000000000000000000000000000000000017
S2140002E2000000000000000000000000000000000007
S2140002461D02F083F302D290AB084B001E3C00085A
S2100002561D02F093590000195900011910
S2070010001000C810
S210000262005A02F60000000C00AA621908
S2080002F6050602668C
S5030013E9
S80400023EBB
```

C++ Support: Appendix H

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, 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 H-1 shows the sample C++ source file.

C++ file:

simple1.cc

```
#include "stream.h"
void main() {
    cout << "hello";
}
```

Figure H-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

_.S0_iostream_init         zerovars:00000000
_.S1                       strings :00000000
___ct__13Iostream_initFv
Iostream_init::Iostream_init()
                                    External
___dt__13Iostream_initFv
Iostream_init::~Iostream_init()
                                    External
___ls__7ostreamFPCc
ostream::operator<<(const char*)
                                    External
___std__simple1_cc_main_      code :0000002C
___sti__simple1_cc_main_      code :0000001C
__main                              External
_adjustfield__3ios
ios::adjustfield
                                    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
            ....
___as__18istream_withassignFP9streambuf
            code        000110DE    stream
istream_withassign::operator=(streambuf*)
            ....
```

The demangled name of the function is listed in the `SYMBOL` column of `___as__18istream_withassignFP9streambuf`.

Symbol Name Length

LNK68K's linkage name limit is 512 characters, which allows for longer mangled names. 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:

- 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)
  ```

- Mangled names have been decoded to show their demangled names. Symbolic names are listed in mangled or demangled pairs.

  Example:

  ```
  ___as__181stream_withassignFP9streambuf
  istream_withassign::operator=(streambuf*)
  ```

- 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
  ```

- Names prefixed by `__std__` are the linkage names for pointers to the C++ static destructors.

  Example:

  ```
  ___std_____stream_filebuf_cxx_last_op_
  ```

- Names prefixed by `__sti__` are the linkage names for pointers to the C++ static constructors.

  Example:

  ```
  ___sti_____stream_filebuf_cxx_last_op_
  ```

- 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

The LIB68K Librarian 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.

HP 64000 Development System Support: Appendix I

Overview

LNK68K can generate 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 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 sets the **debug** and **symbols** flags 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 *Section Types and HP 64000 Symbolic Files* in Chapter 5, *Relocation*, in this 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 a question mark (**?**), a period (**.**), or a dollar sign (**$**). 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 I-1 along with how they are mapped into HP 64000 sections.

  Table I-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 sections containing instructions exist, and no sections containing data definition directives exist, the **DATA** and **COMN** sections will be assigned these "extra" sections containing the instructions.

  If multiple sections containing data definition directives exist, and no sections containing instructions exist, the **PROG** section will be assigned one of these "extra" sections containing the data. Any remaining sections will be mapped into absolute memory 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 to the HP 64700 emulator):

- The HP 64000 uses two special symbols in each routine (**R***label* and **E***label*) 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. The MCC68K Compiler generates these labels automatically 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 ASM68K, the 68000 family monitor program must be uploaded to a host computer, assembled with ASM68K, and linked with 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/Intel assembly language. The modifications required are minor and are summarized in Table I-2.

Table I-2. Modifications Required for the 68000 Monitor

Hewlett-Packard Directive/Instruction	Microtec 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`	`SECTION PROG`