Data General

Software Documentation

Advanced Operating System (AOS) Macroassembler (MASM) Reference Manual

Advanced Operating System (AOS) Macroassembler (MASM) Reference Manual

093-000192-04

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000192

Printed in the United States of America
Revision 04, March, 1984

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

Advanced Operating
System (AOS)
Macroassembler
(MASM)
Reference Manual
093-000192-04

Revision History:

Original Release - June 1976

First Revision - April 1977

Second Revision - June 1978

Third Revision - October 1982

Fourth Revision - March 1984

CONTENT UNCHANGED

The content and change indicators in this revision are unchanged from 093-000192-03. This revision changes only printing and binding details.

Preface

Data General provides two 16-bit macroassembler utilities for ECLIPSE® computers. They are the:

- Advanced Operating System (AOS) Macroassembler (MASM) utility, and the

- Advanced Operating System/Virtual Storage (AOS/VS) 16-bit Macroassembler (MASM16) utility.

This manual describes the use and operation of these utilities. The only difference between the macroassemblers is that AOS MASM runs on 16-bit ECLIPSE computers, while AOS/VS MASM16 runs on 32-bit ECLIPSE computers. Unless we state otherwise in the manual, references to MASM apply to both AOS MASM and AOS/VS MASM16.

The manual assumes you have assembly language programming experience. It also assumes that you are familiar with the assembly language instructions for the 16-bit ECLIPSE computers.

This manual is a reference tool. The following list outlines the contents of each chapter and appendix.

Chapter 1 Introduces the AOS Macroassembler, explains its purpose, and highlights its special capabilities. It also explains the simplest use of MASM.

Chapter 2 Describes the input you pass to the Macroassembler. This chapter is broken into three distinct parts: statement components (e.g., numbers, symbols, expressions), statement format (e.g., labels, comments), and statement types (e.g., instructions, pseudo-ops, assignments).

Chapter 3 Explains how MASM assembles your program. You need not read all of this chapter to use MASM correctly. You should, however, review the sections on memory partitions and relocatability.

Chapter 4 Describes the various forms of output that MASM can produce during an assembly; i.e., object file, assembly listing, cross-reference listing, and error listing.

Chapter 5 Describes the macro facility. It also explains how to use generated numbers and symbols.

Chapter 6 Describes the various types of pseudo-ops you may use in your program.

Chapter 7 Provides detailed descriptions of the AOS MASM pseudo-ops, in alphabetical order.

Chapter 8 Explains the command line that invokes the Macroassembler. This chapter also describes how to link and execute your program, and how to use a permanent symbol table.

Appendix A Contains the ASCII character set.

Appendix B Lists the AOS Macroassembler pseudo-ops.

Appendix C Lists and describes all AOS Macroassembler error codes.

Appendix D Describes the conventions that you should use if you want to assemble AOS assembly-language programs with the AOS/VS Macroassembler.

Suggested Manuals

Many concepts we mention in this manual are documented in greater depth in other Data General publications. In certain instances, you may need to refer to one of the following documents for further information:

* Software Documentation Summary Card (069-000013) lists the names and order numbers for Data General's software documentation.

* The Programmer's Reference, ECLIPSE®-Line Computers manual (015-000024) describes the 16-bit assembly language instruction set for ECLIPSE computers.

* The AOS Link User's Manual (093-000254) describes the AOS Link utility. After assembling your program, you must further process it with Link to produce an executable program file.

* The AOS/VS Link and Library File Editor User's Manual (093-000245) describes the AOS/VS Link and LFE utilities.

- * The AOS Debugger and Disk File Editor User's Manual (093-000195) describes the Debugger utility. The Debugger allows you to examine portions of your program during execution. Use this utility to locate errors in your program.

- * The AOS Library File Editor User's Manual (093-000198) tells you how to use the Library File Editor utility. This utility lets you inspect and change executable program files.

- * The AOS Programmer's Manual (093-000120) documents the AOS system calls and system parameters. System calls are predefined macros that perform commonly used operations.

- * The Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS) (093-000122) tells you how to invoke the Macroassembler utilities.

- * Learning to Use Your Advanced Operating System (069-000018) leads you through sample sessions with AOS, including one with the Macroassembler.

Reader, Please Note:

We use these conventions for command formats in this manual:

COMMAND□required<□optional> ...

Where — Means

COMMAND — You must enter the command as shown.

required — You must enter some argument. Sometimes, we use

$$\left\{\begin{matrix}\text{required}_1\\\text{required}_2\end{matrix}\right\}$$

which means you must enter one of the arguments. Don't enter the braces; they only set off the choice.

<optional> — You have the option of entering this argument. Don't enter the angle brackets; they only set off what's optional.

... — You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

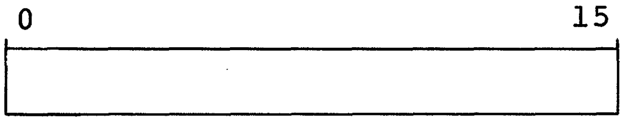
Additionally, we use certain symbols in special ways:

Symbol **Means**

□ The box represents any combination of spaces, horizontal tabs, and one comma. We generally use the box to separate a command and its argument.

<nl> We use the symbol <nl> to represent a statement terminator character. Carriage return form feed, and NEW LINE are all statement terminators.

We use the following format to present bit fields:



Bit 0 is the first bit; we call this the *most significant* bit. Bit 16 is the *least significant* bit in a 16-bit sequence. Each 16-bit segment is called a *word*.

Contacting Data General

* If you have comments on this manual, please use the prepaid Remarks Form that appears after the Index. We want to know what you like and dislike about this manual.

* If you need additional manuals, please use the enclosed TIPS order form (USA only) or contact your Data General sales representative.

* If you experience hardware problems, please notify Data General Systems Engineering.

* Experience software problems -- Please notify your local Data General systems engineer.

End of Preface

Contents

Chapter 1 -- Introduction to the Macroassembler

Developing an Assembly Language Program 1-2
Overview of MASM 1-3
  Macroassembler Input 1-3
  Assembly 1-3
  Macroassembler Output 1-4
The AOS and AOS/VS Macroassemblers 1-4
Special Features of MASM 1-5
Simplest Use of MASM 1-7
  Minimum Necessary Pseudo-Ops 1-7
  Simple MASM Command Line 1-9

Chapter 2 -- Input to the Macroassembler

Character Set 2-1
Source Statements 2-4
Statement Components 2-6
  Terminators and Delimiters 2-7
  Numbers 2-9
    Single-Precision Integers 2-9
    Double-Precision Integers 2-12
    Special Integer-Generating Formats 2-14
    Single-Precision Floating-Point Constants 2-17
  Symbols 2-19
    Symbol Names 2-20
    Symbol Types 2-21
    Permanent Symbols (Pseudo-Ops) 2-22
    Instruction Symbols 2-24
    Redefining Instruction Symbols 2-25
    Macro Symbols 2-26
    Redefining Macro Symbols 2-26
    User Symbols 2-27
    Reserved Symbols 2-27
    Redefining User Symbols 2-27
  Expressions 2-28
    Operators 2-29
    Unary Operators 2-30
    Logical Operators 2-31
    Relational Operators 2-31
    The Bit Alignment Operator (B) 2-32
    Priority of Operators 2-34
    Absolute Versus Relocatable Expressions 2-35
    Special Atoms 2-36
      At Sign (@) 2-36
      Number Sign (#) 2-37
      Asterisks (**) 2-37

Statement Format 2-38
Labels 2-39
Statement Body 2-41
Comments 2-41
Statement Types 2-41
Assembly Language Instructions 2-42
Macros and System Calls 2-43
Pseudo-Ops 2-45
Assignments 2-46
Data 2-47
Defining Assembly Language Instructions 2-48

Chapter 3 -- The Assembly Process

Symbol Interpretation 3-1
Symbol Tables 3-2
Syntax Checking 3-4
Processing Macros and System Calls 3-8
Processing Macro Definitions 3-8
Expanding Macros and System Calls 3-8
Assigning Locations 3-9
Memory 3-9
Location Counter 3-10
Partitions 3-10
Absolute Partition 3-12
ZREL Partition 3-14
NREL Partitions 3-14
Relocatability 3-15
Relocation Bases 3-15
Relocation Bases and Symbols 3-18
Absolute Addresses Versus Absolute Values 3-20
Relocation Bases and Expressions 3-20
Absolute Expressions 3-21
Relocatable Expressions 3-23
Resolving Relocatable Expressions 3-24
Resolving Locations in Memory Reference Instructions . . 3-29
Supplying Both a Displacement and an Index 3-29
Supplying Only a Displacement Value 3-32
Using Literals in Memory Reference Instructions 3-33

Chapter 4 -- Output from the Macroassembler

Object File 4-1
Assembly Listing 4-2
Assembly Listing Control 4-6
Listing Control Pseudo-Ops 4-7
Asterisks (**) 4-7
Cross-Reference Listing 4-8
Error Listing 4-10
Output Function Switches 4-11

Chapter 5 -- Macros and Generated Numbers and Symbols

Macros 5-1
Macro Definition 5-1
  Arguments in Macro Definitions 5-4
Macro Calls 5-5
  Calling Macros without Arguments 5-6
  Calling Macros with Arguments 5-6
  Passing Special Characters and Null Arguments to Macros . 5-8
    Special Characters 5-8
    Null Arguments 5-9
Macro Expansions in Assembly Listings 5-10
Macro-Related Pseudo-Ops 5-12
Loops and Conditionals in Macros 5-12
Macro Examples 5-14
System Calls 5-18
Generated Labels 5-19
Generated Numbers and Symbols 5-20

Chapter 6 -- Types of Pseudo-Ops

Location Counter and Memory Management 6-2
File Termination 6-3
Repetitive and Conditional Assembly 6-3
Macros 6-5
Intermodule Communication 6-5
Listing Control 6-9
Stack Control 6-10
Radix Control 6-11
Text Strings 6-11
Symbol Table 6.12
Symbol Deletion 6-16
Miscellaneous 6-16

Chapter 7 -- Pseudo-Op Descriptions

Coding Aids 7-1
General References 7-1
  (.) 7-3
  .ARGCT 7-4
  .ASYM 7-6
  .BLK 7-7
  .COMM 7-8
  .CSIZ 7-10
  .DALC 7-11
  .DCMR 7-13
  .DEMR 7-15
  .DERA 7-17
  .DEUR 7-19
  .DFLM 7-21
  .DFLS 7-23
  .DIAC 7-25
  .DICD 7-26

.DIMM 7-28
.DIO 7-30
.DIOA 7-32
.DISD 7-34
.DISS 7-36
.DMR 7-38
.DMRA 7-40
.DO 7-42
.DUSR 7-45
.DXOP 7-47
.EJEC 7-49
.END 7-51
.ENDC 7-52
.ENT 7-54
.ENTO 7-56
.EOF 7-58
.EXTD 7-60
.EXTN 7-62
.EXTU 7-64
.FORC 7-65
.GADD 7-66
.GLOC 7-67
.GOTO 7-69
.GREF 7-70
.IFE, .IFG, .IFL, .IFN 7-71
.LMIT 7-73
.LOC 7-75
.LPOOL 7-77
.MACRO 7-79
.MCALL 7-81
.NLIT 7-83
.NOCON 7-84
.NOLOC 7-86
.NOMAC 7-88
.NREL 7-90
.OB 7-92
.PASS 7-94
.PENT 7-95
.POP 7-97
.PTARG 7-98
.PUSH 7-99
.RDX 7-100
.RDXO 7-103
.REV 7-105
.TITL 7-107
.TOP 7-108
.TSK 7-109
.TXT, .TXTE, .TXTF, .TXTO 7-110
.TXTM 7-113
.TXTN 7-115
.XPNG 7-117
.ZREL 7-119

Chapter 8 -- Macroassembler Operating Procedures

MASM Command Line 8-1
Command Line Switches 8-2
Linking and Executing Your Program 8-6
Filenames 8-6
Generic Filenames 8-8
Macroassembler Symbol Tables 8-9
Symbol Resolution 8-9
Permanent Symbol Table 8-10
Building a Permanent Symbol Table 8-11
Specifying a Permanent Symbol Table for an Assembly . . 8-12
Permanent Symbol Table Size 8-12
Symbol Length 8-12

Appendix A -- ASCII Character Set

Appendix B -- Pseudo-Op Summary

Appendix C-- Assembly Error Codes

Addressing Error (A) C-2
Bad Character (B) C-2
Macro Error (C) C-2
Radix (D) C-3
Equivalence Error (E) C-4
Format Error (F) C-4
External/Internal Symbol Error (G) C-4
Input (Parity) Error (I) C-5
Conditional Assembly Error (K) C-5
Location Error (L) C-5
Multiple Definition Error (M) C-5
Number Error (N) C-6
Field Overflow Error (O) C-6
Phase Error (P) C-6
Questionable Line (Q) C-7
Relocation Error (R) C-7
Undefined Symbol Error (U) C-8
Variable Label Error (V) C-8
Text Error (X) C-8

Appendix D -- Assembling 16-Bit Programs with AOS/VS MASM

AOS-Only Pseudo-Ops D-1
Pseudo-Op Interpretation D-1
Macros D-2
Double-Precision Indicator (D) D-2
Literal Values in Memory Reference Instructions D-3
Index Mode Default Conditions D-3
CLI Command-Line Switches D-3

Tables

Table	Caption	
1-1	The ECLIPSE® Macroassemblers	1-5
2-1	Special Characters	2-2
2-2	Statement Terminators	2-8
2-3	Delimiters	2-8
2-4	Digit Representations	2-11
2-5	Sample Integer-Generating Expressions	2-16
2-6	The Four Classes of Symbols	2-22
2-7	Operators	2-29
2-8	Operator Priority Levels	2-34
2-9	Statement Field Delimiters	2-38
2-10	Instructions in Assignment Statements	2-47
2-11	Instruction Definition Pseudo-ops	2-49
3-1	Assigning Addresses Within Partitions	3-16
3-2	Separately Assembled Modules With Similar Partitions .	3-17
3-3	Relocatable Expressions	3-28
3-4	MRI Index Values	3-30
3-5	MRI Displacement Values	3-31
4-1	Object Filename	4-2
4-2	AOS MASM Assembly Listing Fields	4-4
4-3	Location Counter Relocation Symbols	4-5
4-4	Data Field Relocation Symbols	4-6
4-5	Assembly Listing Control Pseudo-Ops	4-7
4-6	Cross-Reference Assignment Mnemonics	4-9
4-7	AOS Macroassembler Output Function Switches	4-12
5-1	Generated Symbols in Source and Listings	5-21

6-1 Location Counter and Memory Management Pseudo-Ops . . 6-2
6-2 File Termination Pseudo-Ops 6-3
6-3 Repetitive Assembly and Conditional Pseudo-Ops . . . 6-4
6-4 Macro-Related Pseudo-Ops 6-5
6-5 Intermodule Communication Pseudo-Ops 6-6
6-6 Listing Control Pseudo-Ops 6-9
6-7 Stack Control Pseudo-Ops 6-10
6-8 Radix Control Pseudo-Ops 6-11
6-9 Text String Pseudo-Ops 6-12
6-10 Symbol Table Pseudo-Ops 6-13
6-11 Symbol Deletion Pseudo-Op 6-16
6-12 Miscellaneous Pseudo-Ops 6-17

7-1 Abbreviations 7-2
7-2 DUSR Assignments Versus Simple Assignments 7-46
7-3 Numeric Representations in Various Bases 7-101

8-1 AOS MASM Function Switches 8-3
8-2 MASM Argument Switch 8-5
8-3 AOS Filename Extensions 8-7
8-4 Object Filename 8-8

B-1 Pseudo-Op Summary B-1

Illustrations

Figure **Caption**

2-1 The Source Statement 2-6

3-1 Resolving Symbols 3-3
3-2 Processing Source Statements 3-6
3-3 Sorting Code into Memory Partitions 3-11
3-4 Linking Modules with Similar Partitions 3-17

4-1 Sample Assembly Listing 4-3
4-2 Sample Cross-Reference Listing 4-9

5-1 Macro Listing 5-10

Chapter 1
Introduction to the Macroassembler

A language is a set of representations, conventions, and rules that convey information in a well-defined way. At the lowest level, a computer language consists of numeric codes that represent computer hardware operations. The computer can readily understand this numeric language, called machine language.

As an example, the following machine language code represents a jump instruction to address 377_8:

```
     000377
```

000000 is the base value of the jump instruction and 377 is the location.

Coding a program that uses the many ECLIPSE® 16-bit machine language instruction numbers would be a time-consuming, cumbersome process. Thus, for programming convenience, we assign each machine language instruction a symbolic name that has significance to us.

For example, the machine language instruction 000000 receives the name JMP and our jump statement is now

```
     JMP            377
```

This instruction is much simpler to read because the mnemonic JMP is similar to the English word, jump. The mnemonic implies the operation that the corresponding machine-level instruction performs.

We may further simplify machine language by using symbols to represent locations as well as instructions. In the last example, we could assign location 377 the symbolic name LOC1. Our instruction would now be

```
     JMP            LOC1
```

This is relatively easy to read and understand; i.e., "Jump to location LOC1."

The programming language that consists of symbols instead of numbers is called assembly language. In general, each assembly-language instruction corresponds to one machine-language instruction. Thus, assembly language provides the same set of operations as machine language but is much simpler to use.

Unlike machine language, assembly language is not understood by the computer. Thus, you must translate your symbolic assembly language program into its machine language equivalent. The Macroassembler, or MASM, is the program that performs this translation operation.

Developing an Assembly Language Program

Before discussing the operation of the Macroassembler, we should briefly review the five steps necessary to produce, execute, and debug an assembly language program.

1. Writing and entering your program -- The first step in producing your program is, of course, writing its assembly language instructions. Next, you must enter the program into the computer. Normally, you will enter your program from a console using one of Data General's text editors. Your assembly language program is called a source module and, if stored on disk, it resides in a source file. (By convention, source filenames end with the .SR extension.)

2. Assembling your program -- After you enter your program, invoke the Macroassembler to translate your symbolic assembly language source module into its numeric machine language equivalent. The Macroassembler places this machine language program, called the object module, into a new file, called the object file. (The Macroassembler always ends object filenames with the .OB extension.) If the Macroassembler detects an error in your source module, you must edit and reassemble your source file before continuing the program development process.

3. Linking your program -- After a successful assembly, you must use the Link utility to produce an executable program. Link pulls your object modules apart and rearranges them into an image of your program as it will appear in memory during execution. Link stores this image in a program file. (Link always ends program filenames with the .PR extension.) As with the Macroassembler, the Link utility may detect errors in your program. If so, you must edit, reassemble, and relink your program to correct those errors.

4. Executing your program -- After you successfully link your program, you may execute it by typing

   ```
   XEQ     program-filename<nl>
   ```

   at your console. If your program runs smoothly and performs the appropriate actions, the program development process is complete.

5. Debugging your program -- Often, your program does not run correctly the first time. It may cause a runtime error, or not perform the desired operations.

In either case, you must debug your program (i.e., remove the errors or "bugs"). Sometimes you can readily detect the problem. In these cases, simply correct the source module. Then, reassemble and relink your program. If you cannot locate the error in your source module, you may use the AOS Debugger utility to examine portions of your program during execution. Again, after locating the error, you must edit, reassemble, and relink your program.

The above outline is meant as a brief overview of assembly language programming. The rest of this manual focuses on step 2, the assembly process. Refer to the Preface for a list of manuals that describe the other programming steps.

Overview of MASM

The following sections of this manual briefly discuss the input you pass to the Macroassembler, the assembly process, and the output that MASM produces. Chapters 2, 3, and 4 will discuss these topics in much greater detail.

Macroassembler Input

The source module you pass to the Macroassembler consists of characters grouped into a series of source statements. In general, your source statements may

- perform operations at execution time,
- contain data, or
- direct the operation of the Macroassembler.

Assembly

The Macroassembler reads the statements in your source module twice. During these two passes through your program, MASM

- interprets symbols,
- checks the syntax of your source statements,
- resolves memory locations, and
- expands macros and system calls (we describe these later in this chapter).

Macroassembler Output

The Macroassembler can produce five types of output:

* Object file

* Assembly listing

* Cross-reference listing

* Error listing

* Permanent symbol table

The object file holds the machine language version of your source module.

The assembly listing allows you to compare your input with the Macroassembler's output. This listing contains your original source statements plus the numeric machine language values produced by MASM.

The cross-reference listing provides an alphabetic list of the symbols you use in your program and their values.

The error listing contains information about all statements in your source module that cause assembly errors.

The permanent symbol table holds symbol definitions for use in future assemblies. If you use a permanent symbol table, you need not redefine frequently used symbols for each assembly. The Advanced Operating System (AOS) software package provides a standard permanent symbol table for your use. By default, MASM uses the standard table for each assembly.

The AOS and AOS/VS Macroassemblers

Data General provides three Macroassembler (MASM) programs for the ECLIPSE computers: AOS MASM, AOS/VS MASM16, and AOS/VS MASM.

AOS MASM runs on 16-bit ECLIPSE computers. It assembles 16-bit (AOS) assembly language source files to produce object files for 16-bit or 32-bit ECLIPSE computers. AOS/VS MASM16 provides the same functions as AOS MASM. However, AOS/VS MASM16 runs on 32-bit ECLIPSE computers only. This manual describes the AOS and AOS/VS MASM16 Macroassemblers.

The AOS/VS MASM utility assembles 32-bit (AOS/VS) source files to produce object files that run only on the 32-bit series ECLIPSE computers. The AOS/VS Macroassembler is described in the AOS/VS Macroassembler Reference Manual.

Table 1-1 summarizes the Macroassembler utilities.

Table 1-1. The ECLIPSE Macroassemblers

Macroassembler	Source Programs	Description
AOS MASM	16-bit ECLIPSE assembly language instructions	Runs on 16-bit ECLIPSE computers. Produces object code that can be linked and run under either AOS or AOS/VS
AOS/VS MASM16	16-bit ECLIPSE assembly language instructions	Runs on 32-bit ECLIPSE computers. Produces object code that can be linked and run under either AOS or AOS/VS
AOS/VS MASM	32-bit ECLIPSE assembly language instructions	Runs on 32-bit ECLIPSE computers. Produces object code that must be linked and run under AOS/VS only

To take full advantage of the 32-bit ECLIPSE computer's performance, you must assemble your programs with the AOS/VS Macroassembler. Appendix D describes the program-visible differences between the AOS and AOS/VS Macroassemblers.

Special Features of MASM

The Macroassembler's primary function is to translate your symbolic assembly language program into its numeric machine language equivalent. During this process, the Macroassembler can perform a variety of operations that increase your programming power and, at the same time, simplify your source code.

The following is a partial list of these special Macroassembler features. Detailed descriptions appear elsewhere in this manual.

Symbolic location names

As mentioned earlier, the Macroassembler allows you to assign symbolic names, called labels, to memory locations

Number representations

You have three different internal number representations at your disposal (single-precision integer, double-precision integer, and floating-point)

Expression evaluation

The Macroassembler provides arithmetic, logical, and relational operators that you may use for number manipulation

Memory management

The Macroassembler can either assign absolute addresses to your object code or assign relocatable addresses that are resolved at link time

Repetitive assembly

You may direct MASM to assemble a series of source statements a specified number of times; that is, you may implement a DO loop at assembly time

Conditional assembly

The Macroassembler allows you to conditionally assemble or bypass a section of source code based on the evaluation of an expression; in other words, you may implement an IF-THEN-ELSE structure at assembly time

Text string storage

You may direct MASM to store in memory the ASCII codes for any string of characters

Radix control

You may alter the radix (base) for numeric input to and output from the Macroassembler

Assembler stack

The Macroassembler provides a pushdown stack that operates at assembly time

Intermodule communication

You may assemble source modules separately and then link them together into a single program file. These separately assembled modules can share data and symbol definitions

Macros

The macro facility allows you to assign a symbolic name to a series of source statements. Then, each time you want to insert that source code, simply enter the assigned name. At assembly time, the Macroassembler correctly expands the macro name to the original source statements

Instruction definition

The Macroassembler lets you define your own instructions, using the formats of the standard 16-bit ECLIPSE instructions. The Macroassembler is supplied with files that define the standard ECLIPSE instructions. You can add to the instruction definitions in these files, or replace the standard definitions with your own.

The Macroassembler supplies these and other programming tools to help you develop programs. Note that the Link utility and the AOS system calls (predefined macros) provide further assembly language programming control. These two subjects are, however, outside the realm of this manual. (See the Preface for a list of relevant manuals.)

Simplest Use of MASM

In certain situations, you may not wish to use the many advanced operations performed by the Macroassembler. In the following two sections of this manual, we ignore these special features and describe the simplest way for you to assemble your program.

Minimum Necessary Pseudo-Ops

Pseudo-op directives are source statements that direct the assembly process. Your program never executes them; rather, MASM evaluates them and performs the appropriate operations at assembly time.

Chapters 6 and 7 describe the various pseudo-ops in detail. Refer to those chapters for clarification of any points we mention in the following discussion.

Generally, you should use at least three pseudo-ops in each source module:

- .TITLE
- a location directive
- .END

.TITLE places a name in the object module for later use by the Link utility. MASM repeats this name at the top of each page in your assembly listing.

The syntax for using .TITLE in your source module is

.TITLE name

where:

name is the name you wish to assign to the object module

If you do not include a .TITLE pseudo-op in your source, MASM supplies the title .MAIN, by default.

The second pseudo-op statement in your source module should tell the Macroassembler where in memory your program will reside at execution time. In most cases, your program will be relocatable and can reside anywhere in unshared memory. The pseudo-op statement that conveys this to the Macroassembler is

.NREL

If your program is relocatable but must reside in lower page zero (i.e., below location 400_8), use the .ZREL pseudo-op. Use .LOC if you want your program to reside at a specific memory address.

If you do not include a location directive in your source module, the Macroassembler begins assigning addresses at absolute location 0. Normally, you do not want to place code below address 50_8. These locations are used by the operating system and hardware.

The third pseudo-op you should include in your source module is .END. The Macroassembler does not process any source code that follows .END, so this should be the last statement in your program.

When you issue the .END pseudo-op, you must supply a symbol that specifies a starting address for execution of your program.

Thus, the syntax for using .END is

```
        .END      label
```

where:

label is a symbolic name for the address where program execution will begin

In sum, the general format for your source module should be

```
        .TITLE          name
        .NREL
label:                .
                      .
                      .
                      (your assembly
                      language program)
                      .
                      .
                      .
        .END            label
```

Again, Chapters 6 and 7 describe all pseudo-ops in detail.

We should point out that your source module should not contain assembly language I/O (Input/Output) instructions if you intend to run your program under an operating system (e.g., AOS). Instead, you must use I/O system calls (predefined macros that handle input and output for you). However, you can use I/O instructions with the operating system's IDEF facility. The AOS Programmer's Manual describes the I/O system calls and IDEF facility.

Simple MASM Command Line

To assemble your source module, enter the following CLI command:

```
        XEQ  { MASM   }  sourcefile<nl>
             { MASM16 }
```

where:

XEQ is a CLI command that executes a program

MASM is the name of the Macroassembler program (without the .PR extension). Use MASM16 if you are assembling 16-bit programs under AOS/VS.

sourcefile is the name of the file that contains your source module (the .SR extension is optional)

During assembly, MASM creates a file to hold your object module. MASM assigns this file the name of the source file (i.e., **sourcefile**), without the .SR extension (if any) and with the .OB extension.

If the Macroassembler detects any errors in your source module, it reports them to the generic file **@OUTPUT.** When you are using MASM interactively, **@OUTPUT** is your console.

As an example, suppose your source module resides in file PROG1.SR. The command that assembles this file is

```
        ( MASM   )
XEQ    <          > PROG1<nl>
        ( MASM16 )
```

Note that you need not include the .SR extension on the name of the source file.

When you issue this command, the Macroassembler creates file PROG1.OB. At the end of the assembly, this file contains your object module.

The above discussion summarizes the simplest way for you to assemble a program. However, the Macroassembler provides you with many options at assembly time. For example, you may

* produce an assembly listing,

* send errors to a specific file (i.e., besides **@OUTPUT**),

* specify the name for your object file, and

* suppress production of the object file.

Chapter 8 describes these and other features of the MASM command line in detail. That chapter also provides information on linking and executing your program.

End of Chapter

Chapter 2
Input to the Macroassembler

The input you pass to the Macroassembler is in the form of one or more assembly language source modules. In this chapter, we discuss the different elements that make up a source module.

In most cases, you enter your source module with one of Data General's text editors. When you name a source file (i.e., a file that contains a source module), add the .SR extension to the end of the filename (e.g., source file PROG.SR). Chapter 8 describes the Macroassembler's naming conventions for files. Chapter 8 also explains the Macroassembler operating procedures.

Character Set

Each source module consists of a string of ASCII characters. The AOS Macroassembler allows you to use the following characters in a source module:

* Uppercase and lowercase alphabetic characters: A through Z and a through z. By default, the Macroassembler is not case sensitive (e.g., the symbols 'START' and 'start' are the same).

* Numerals: 0 through 9

* Dollar sign and question mark: $?

* Format control and end-of-line characters: carriage return, form feed, NEW LINE, space, horizontal tab.

* Special characters:

"	'	!	&	+	*	
/	-	<	>	=	@	#
%	,	_	.	:	;	^
\	E	B	**	()	[]	D

These characters have special meanings to the Macroassembler. Table 2-1 lists the meaning of each special character and provides references for more information.

The Macroassembler unconditionally ignores null characters (ASCII 000$_8$). Do not use the following characters in your source module: delete (ASCII 177$_8$), control characters, or characters with the parity bit set to 1. If the Macroassembler encounters one of these, it returns a bad character (B) error and ignores the illegal character.

Appendix A lists the octal codes for each ASCII character.

Table 2-1. Special Characters

Character	Meaning	Reference
:(colon)	Follows all labels	"Labels" - Chapter 2
;(semicolon)	Precedes all comments	"Comments" - Chapter 2
.(period)	A permanent symbol with the value and relocation property of the current location counter	(.) pseudo-op description - Chapter 7
	Indicates a decimal integer or a floating-point constant	"Numbers" - Chapter 2
	May appear in symbol names	"Symbol Names" - Chapter 2
,(comma)	Delimits arguments	"Delimiters" - Chapter 2
+(plus sign)	Addition operator	"Operators" - Chapter 2
	Unary operator indicating a positive value	"Unary Operators" - Chapter 2
-(minus sign)	Subtraction operator	"Operators" - Chapter 2
	Unary operator indicating a negative value	"Unary Operators" - Chapter 2
*(asterisk)	Multiplication operator	"Operators" - Chapter 2
/(slash)	Division operator	"Operators" - Chapter 2
B(capital B)	Single-precision bit alignment operator (16 bits)	"Bit Alignment Operator" - Chapter 2

(continues)

Table 2-1. Special Characters

Character	Meaning	Reference
&(ampersand)	Logical AND operator	"Logical Operators" - Chapter 2
!(exclamation point)	Logical OR operator	"Logical Operators" - Chapter 2
>(greater than)	Relational operator	"Relational Operators" - Chapter 2
<(less than)	Relational operator	"Relational Operators" - Chapter 2
=(equals sign)	Assigns a value to a symbol	"Assignments" - Chapter 2
	Combines with other characters to form relational operators	"Relational Operators" - Chapter 2
'(apostrophe)	Converts two ASCII characters to their octal values	"Special Integer-Generating Formats" - Chapter 2
"(quotation mark)	Converts an ASCII character to its octal value	"Special Integer-Generating Formats" - Chapter 2
^(uparrow)	Identifies formal arguments in a macro definition string	"Arguments in Macro Definitions" - Chapter 5
%(percent)	Terminates a macro definition string	"Macro Definition" - Chapter 5
_(underscore)	Directs the assembler to ignore the special meaning of a character that appears in a macro definition string	"Macro Definition"-Chapter 5
	May appear in symbol names	"Symbol Names" - Chapter 2
\(backslash)	Generates numbers and symbols	"Generated Numbers and Symbols" - Chapter 5

(continued)

Table 2-1. Special Characters

Character	Meaning	Reference
D(capital D)	Double-precision integer indicator	"Double Precision Integers"- Chapter 2
E(capital E)	Exponential notation indicator	"Single-Precision Floating-Point Constants" - Chapter 2
()(parentheses)	May surround a number, symbol, or expression to alter operator priority	"Priority of Operators" - Chapter 2
[](square brackets)	May enclose arguments in a macro call	"Macro Calls" - Chapter 5
**(double asterisks)	Suppress listing of the source line	"Asterisks (**)" - Chapter 4
@(at sign)	Indirect addressing indicator; directs the assembler to place a 1 in the indirect addressing bit	"At Sign (@)" - Chapter 2
#(number sign)	No-load indicator; directs the assembler to place a 1 in the no-load bit	"Number Sign (#)" - Chapter 2

(concluded)

Source Statements

An assembly language source module consists of a series of source lines or statements. A source statement is a sequence of ASCII characters terminated by an end-of-line character (also called a statement terminator). Carriage return, form feed, and NEW LINE characters all act as statement terminators. In this manual, we use the symbol <nl> to represent statement terminators.

Examples of three source statements are

```
                325 <nl>
                LDA   5,LOCX <nl>
BEGIN:         .ZREL            ;LOWER PAGE ZERO RELOCATABLE<nl>
```

A source statement may not be more than 132 characters in length. If a statement is more than 132 characters long, the Macroassembler truncates the line and returns an error.

There are five different types of source statements. All consist of the same basic components and all must conform to the same general format. Thus, we focus on three distinct topics in the remainder of this chapter:

* statement components

* statement format

* statement types

Figure 2-1 outlines the major subjects under each of these three topics. The organization of the following presentation closely conforms to that figure. Please note that the same information may appear in several sections of this chapter, if appropriate.

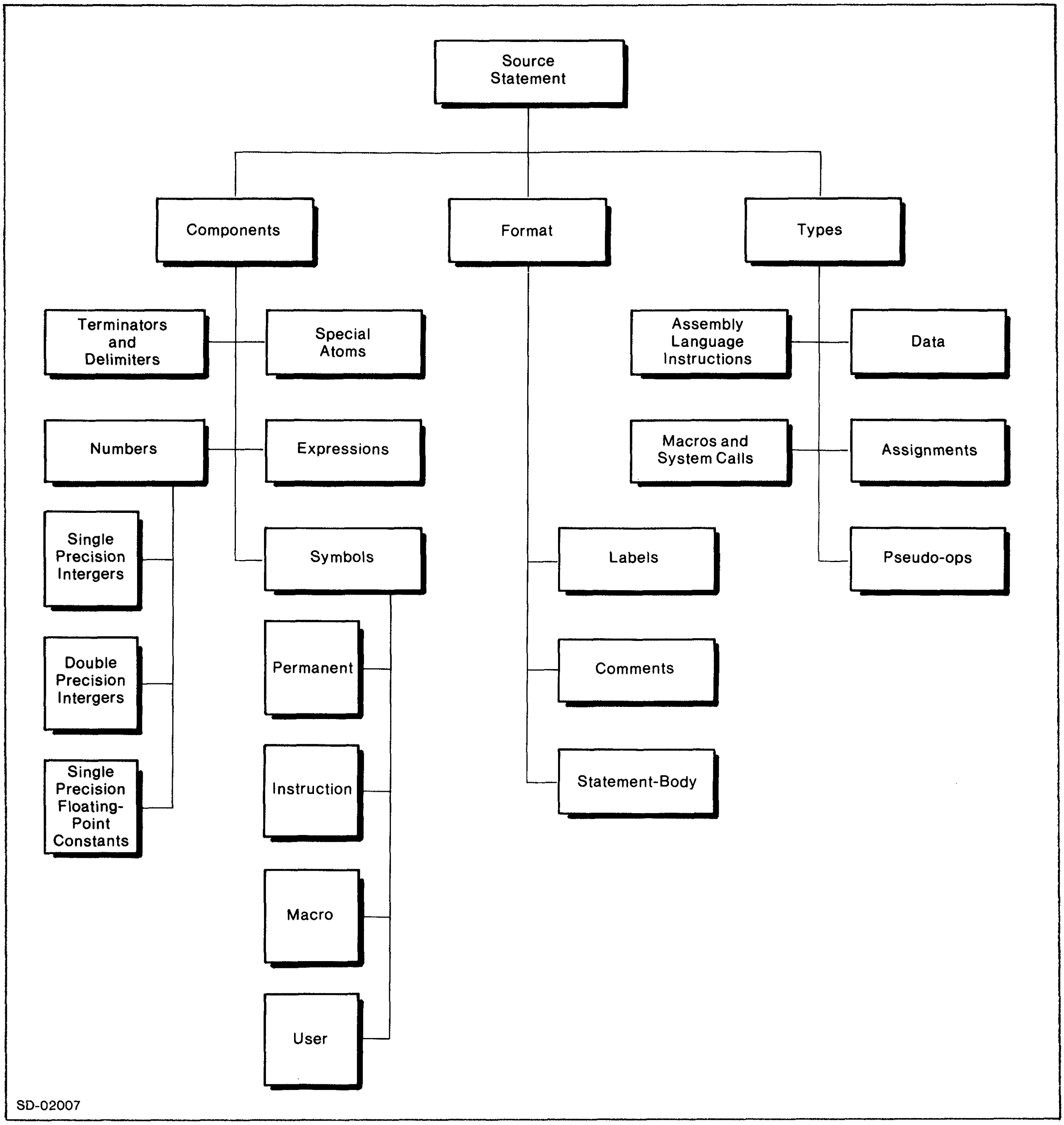


Figure 2-1. The Source Statement

Statement Components

A source statement consists of one or more syntactic units, called atoms. Each atom is a string of one or more ASCII characters that the Macroassembler views as a single entity.

There are four types of atoms:

- Terminals and delimiters
- Numbers
- Symbols
- Special atoms

In many cases, you combine these atoms to form expressions. An *expression* is a series of symbols and/or numbers separated by operators. (As we shall see, an operator is a delimiter atom.) For example, X+2 is an expression that consists of a symbol atom and a number atom joined by the operator +.

Though an expression is not an atom, the Macroassembler often views an expression as a single entity. For example, you may supply an expression as a single argument to an instruction, pseudo-op, or macro call. In the instruction

```
LDA     0,X+2
```

the Macroassembler treats X+2 as a single entity, distinct from the symbol LDA and the number 0.

Thus, when discussing the major components of source statements, we include expressions along with the four types of atoms. Our list of basic statement components is now

- terminals and delimiters,
- numbers,
- symbols,
- expressions, and
- special atoms.

In the following sections of this manual, we describe each of these statement components in detail.

Terminators and Delimiters

Terminators are characters that separate the source statements in your module. Table 2-2 lists the terminators, also called *end-of-line* characters.

Table 2-2. Statement Terminators

Character	ASCII Code (octal)
Carriage return	015
Form feed	014
NEW LINE	012

In this manual, we represent all terminators with <nl>.

Delimiters are characters that separate numbers, symbols, and expressions from each other within a single source statement. Table 2-3 lists and describes the various delimiters.

Table 2-3. Delimiters

Symbol	Description
}(box)	Any combination of spaces, horizontal tabs, and one comma.
=	Assigns a value to the symbol preceding this sign.
:	The symbol preceding this character is a label.
;	Indicates the beginning of a comment string.
+ - * / B	Arithmetic operators.
== > < <= >= <>	Relational operators.
& !	Logical operators.
()	May enclose a number, symbol, or expression.
[]	May enclose the arguments in a macro call.
%	Terminates a macro definition string.
_	Directs the Macroassembler to ignore the special meaning of a character that appears in a macro definition string.

Numbers

The following discussion explains the various number representations you may use in a source module. Number is a general term that refers to integers (whole numbers) and floating-point constants (fractions and exponential values).

The Macroassembler lets you use three different types of number representations:

* single-precision integer, stored in one word (16 bits)

* double-precision integer, stored in two words (32 bits)

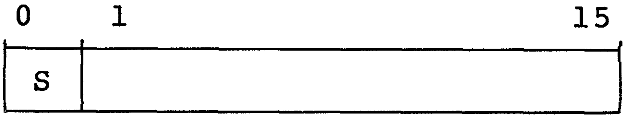
* single-precision floating-point constant, stored in two words (32 bits)

Single-precision integers may appear in expressions and data statements. Double-precision integers and floating-point numbers may appear only in data statements.

Single-Precision Integers

The Macroassembler represents single-precision integers as single 16-bit words in the range 0 to 65,535$_{10}$ (0 to 177,777$_{8}$). You can use twos-complement notation to represent any signed integer in the range -32,768$_{10}$ to +32,767$_{10}$.

The first bit (bit 0) is the sign bit. If that bit equals 0, the integer is positive; if it equals 1, the integer is negative.



Single-Precision Integer Representation

The format of a single-precision integer in your source module is

<sign>**d**<d...><.>**break**

where:

sign is the integer's sign; use - for negative numbers and + for positive numbers. If you do not supply a sign, the Macroassembler assumes that the integer is positive.

d is a digit in the range of the current input radix; the first digit must be in the range 0 through 9.

. is an optional decimal point. The Macroassembler interprets the integer as decimal (base 10) if you supply the decimal point.

break terminates the integer. The **break** character may be any delimiter or terminator (see "Terminators and Delimiters" earlier in this chapter).

If a decimal point precedes the **break** character, the Macroassembler evaluates the integer as decimal. If you omit the decimal point, the Macroassembler evaluates the integer in the current input radix. You may set the input radix to any base from 2 to 20 (see the .RDX pseudo-op in Chapter 7). Table 2-4 shows the digit representations for the various bases.

Table 2-4. Digit Representations

Radix (base)	Highest Digit	Highest Digit's Decimal Value
2	1	1
3	2	2
4	3	3
5	4	4
6	5	5
7	6	6
8	7	7
9	8	8
10	9	9
11	A	10
12	B	11
13	C	12
14	D	13
15	E	14
16	F	15
17	G	16
18	H	17
19	I	18
20	J	19

When you select a radix of 11 or greater, your integers may contain letters that represent digits. For example, in base 16, the number 2F represents the value 47_{10}.

If the first digit of an integer starts with a letter, you must precede that integer with the digit 0. Otherwise, the Macroassembler cannot distinguish the integer from a symbol.

The following examples of legal hexadecimal (base 16) integers help clarify this rule:

```
0F
0A45
6A9
333
45B
0B2F
```

You may end a single-precision integer with any operator, delimiter, or terminator. All operators are delimiters (we discuss operators later in this chapter).

Note that the bit alignment operator B is an exception to the above rule. If you are using an input radix of 12 or greater, the Macroassembler interprets B as a digit. If you want the Macroassembler to interpret B as the bit alignment operator, place the preceding operand inside parentheses. For example,

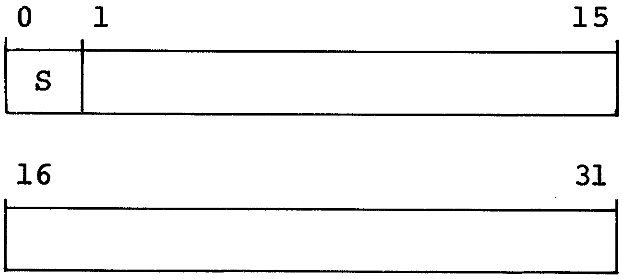
```
.RDX        16          ;Input radix equals 16.
49B3                    ;The Macroassembler interprets B as
                        ;a hexadecimal digit.
(49)B3                  ;The Macroassembler interprets B as
                        ;the bit alignment operator.
```

Refer to "Expressions" in this chapter for a description of the bit alignment operator B.

Double-Precision Integers

The Macroassembler represents double-precision integers in two consecutive words of memory (32 bits). Using twos-complement notation, you can represent any signed integer from $-2,147,483,648_{10}$ to $+2,147,483,647_{10}$. Unsigned double-precision integers may range from 0 to $4,294,967,295_{10}$.

The first bit of the first word (bit 0) is the sign bit. If that bit equals 0, the integer is positive; if it equals 1, then the integer is negative.



Double-Precision
Integer Representation

The general format for a double-precision integer in a source module is

<sign>**d**<d...><.>**D**□**break**

where:

sign — is the integer's sign; use - for negative numbers and + for positive numbers. If you do not supply a sign, the Macroassembler assumes that the integer is positive.

d — is a digit in the range of the current input radix; the first digit must be in the range 0 through 9.

. — is an optional decimal point. The Macroassembler interprets the integer in base 10 if you supply the decimal point.

D — tells the Macroassembler to store the integer in double-precision format

break — terminates the integer. The **break** character may be any delimiter or terminator (see "Terminators and Delimiters" earlier in this chapter).

According to this definition, all of the following are legal double-precision integers:

25D   1320.D   -1D   +241D   -177777D

The following conventions apply to the use of double-precision integers:

* If a decimal point precedes the D character, the Macroassembler interprets that integer as decimal.

* The radix of a double-precision integer can be in the range 2-20. If the radix is greater than or equal to 14, the letter D will be interpreted as a digit. However, you can force the Macroassembler to interpret D as the double-precision indicator by placing an underscore character (_) ahead of the D.

* The first digit in each integer must be in the range 0 through 9. If the input radix is 11 or greater, your integer may contain letters (e.g., $3F_{16}$). If the first digit of a number is a letter, precede that letter with a zero (i.e., use 0F5 instead of F5).

* If the input radix is 12 or greater, an operand that precedes the bit alignment operator B must be within parentheses (e.g., (29)B5_D).

The following listing shows how the Macroassembler assembles double-precision integers.

Input	**Assembled Value**	**Radix**
200000D	000001 000000	8
2000000.D	000036 102200	10
-12767D	177777 165011	8
12D	000455	8
12_D	000000 000022	16
0F31_D	000000 007461	16

Special Integer-Generating Formats

Two special input formats convert ASCII characters to integers.

The first format converts a single ASCII character to its 8-bit octal value. The input format is

"a

where:

" is a quotation mark that directs the Macroassembler to store the ASCII code for the following character.

a represents any legal ASCII character, except rubout (1778) or null (000). See "Character Set" at the beginning of this chapter for a list of the legal characters.

The Macroassembler interprets only the character immediately following the quotation mark. If you include extra characters, MASM assembles the first one correctly and returns an error for the subsequent characters.

A few examples follow to illustrate the use of the quotation mark.

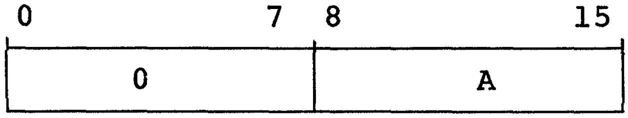
Input	**Octal Value**
"5	65
"A	101
"%	45
"%T	45 (the character T generates an error)

You may also use the quotation mark format as part of an expression. The following examples illustrate this:

Input	**Octal Value**
"A+4	101+4
"C*5	103*5
"#-"%	43-45

In every case, <nl> assembles the octal value for the terminator character and also terminates that source line. Thus, if you place a quotation mark before a NEW LINE character, the Macroassembler assembles the octal value for NEW LINE (i.e., 012_8). The Macroassembler also interprets that same NEW LINE character as a terminator for that source statement.

The Macroassembler packs the value generated by this format in the rightmost byte of a word in memory (i.e., in the word's least significant 8 bits). Bit eight will be set to zero. For example, the Macroassembler stores "A as follows:



The second special integer-generating format converts up to two ASCII characters into an integer. The format is

'string'

where:

' is an apostrophe; MASM requires you to enclose the ASCII characters in apostrophes

string consists of any number of ASCII characters; the Macroassembler uses only the first two characters of the string

The Macroassembler packs the octal values of **string's** first two characters from left to right in a 16-bit word. For example, the Macroassembler stores 'A' as follows

0 7	8 15
A	0

The Macroassembler stores both 'AB' and 'ABCD' as

0 7	8 15
A	B

Two apostrophes without an intervening character string generate an integer containing all zeros (i.e., absolute zero). A newline entered before the second apostrophe terminates the string format.

You may use the two special integer-generating formats wherever the Macroassembler allows you to use integers. Table 2-5 shows some simple expressions that use the special formats.

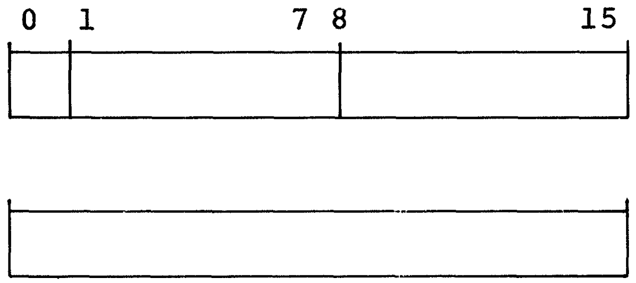
Table 2-5. Sample Integer-Generating Expressions

Source	Octal Value
"A	101
'AB'	40502
'BA'	41101
''	0
+5-2	3
'B'+5	41005
' A'	20101
"A+'A'	40501
'AA'	40501
'AABCD'	40501

Single-Precision Floating-Point Constants

Floating-point constants represent fractional and exponential values. We refer to these numbers as constants because they cannot appear in expressions or assignments. They may appear only in data statements.

The Macroassembler uses two contiguous words of memory (32 bits) to represent a single-precision floating-point number.



Single-Precision Floating-Point Constant

Bit 0 is the sign bit. If that bit equals 0, the number is positive; if it equals 1, the number is negative.

Exponent is the integer exponent of 16, expressed in excess-64_{10} (100_8) notation. The Macroassembler represents exponents from -64_{10} to $+63_{10}$ with their binary equivalents from 0 to 127_{10} (0 to 177_8). The Macroassembler represents a zero exponent as 100_8.

The Macroassembler represents the mantissa as a 24-bit binary fraction. You may view the mantissa as six 4-bit hexadecimal digits. The range of the mantissa's magnitude is

$$16^{-1} <= \text{mantissa} <= (1-16^{6})$$

You may obtain the negative form of a floating-point number by complementing bit 0 (i.e., from 0 to 1, or from 1 to 0). The exponent and mantissa remain the same.

The magnitude of a floating-point constant is

$$(16^{-1})*(16^{-64}) <= \text{floating-point constant} <= (1-16^{6})*(16^{63})$$

which is approximately

$$5.4*10^{-79} <= \text{floating-point constant} <= 7.2*10^{75}$$

The Macroassembler normalizes all nonzero floating-point numbers. A floating-point number is normalized if the fraction (mantissa) is greater than or equal to 1/16 and less than 1. In other words, the binary representation of a normalized number has a 1 in one of the first four bits (8-11) of the mantissa. For example, if you specify the number 65.32, the Macroassembler converts it to $.6532*10^{2}$.

Much of the floating-point number source format is optional. The minimum format is one digit, followed by either a decimal point or the letter E, followed by another digit. Thus, the minimum floating-point number format is

d $\left\{\begin{matrix}\mathbf{E}\\ \bullet\end{matrix}\right\}$ **d□break**

where:

d is a single digit in the range 0 through 9.

For example, 3.5 and 6E2 are both floating-point constants.

The complete source format for a single-precision floating-point number is

<sign>**d**<d...>**.d**<d...><**E**<sign>d<d>>**break**

or

<sign>**d**<d...>**E**<sign>d<d>**break**

where:

sign indicates the sign of a value (positive or negative) and is one of the following characters: + or -. If the sign appears before the number, then it defines the sign of that number. If a sign character appears after the letter E, then it defines the exponent's sign. If you do not supply a sign, the Macroassembler assumes that the value is positive.

d is a digit in the range 0 through 9. The Macroassembler always interprets the mantissa and exponent as decimal (e.g., 26.5 equals $.265*10^2$ regardless of the current input radix).

. is an optional decimal point. If you include a decimal point but do not follow that point with either a digit or the letter E, the Macroassembler stores the value as an integer, not a floating-point number.

E indicates floating-point number representation. You must follow the E with one or two digits representing the value of the exponent.

break terminates the floating-point number. The **break** character may be any delimiter or terminal (typically } ; or <nl>).

You may format the same floating-point number with the letter **E**, a decimal point, or both. For example,

Floating-Point Constant	**Assembled Value**
254.33	041376 052173
254.33E0	041376 052173
25433E-02	041376 052173
25433E-2	041376 052173
2543.3E-1	041376 052173
0.25433E03	041376 052173

The two octal numbers under the heading "Assembled Value" depict the two 16-bit words that represent the floating-point constant's value.

If the current input radix is 15 or greater, the Macroassembler may interpret the letter E as a digit rather than the floating-point number indicator. To avoid ambiguity, precede the exponential E with a period (.) when representing a floating-point constant. For example,

```
.RDX    16      ;Input radix is 16.
-5E3            ;E is a hexadecimal digit and -5E3 represents
                ;an integer.
-5.E3           ;E indicates floating-point number
                ;representation (i.e., -5*10$^3$).
```

The following examples show floating-point numbers and the corresponding values that the Macroassembler stores.

Floating-Point Constant	**Assembled Value**
1.0	040420 000000
3.1415926	040462 041767
-1E0	140420 000000
+5.0E-1	040200 000000
+273.0E0	041421 010000
0.33E2	041041 000000

Symbols

Each assembly language source program you write will contain ASCII character strings called symbols. Each symbol represents a binary number. One function of the Macroassembler is to translate the symbols in your source program into their binary machine language equivalents. Assembly language instructions (e.g., LDA), pseudo-ops (e.g., .NREL), system calls (e.g., ?READ), and labels are all symbols).

Symbol Names

Every symbol in your source program must conform to the following syntax:

a<b>...**break**

where:

a is the first character of the symbol and may be any upper- or lowercase letter (A - Z,a - z), period (.), dollar sign ($), or question mark (?)

b represents succeeding characters in the symbol and can include upper- and lowercase letters (A - Z, a - z), numbers (0 - 9), period (.), dollar sign ($), question mark (?), and underscore (_)

break terminates the symbol; a **break** character may be any delimiter or terminator (see "Terminators and Delimiters" earlier in this chapter)

According to these rules, the following character strings are all legal symbols:

```
.START      B12      EXIT_1      $Z
```

The following strings are all illegal symbols; the first two strings do not begin with a letter, period, question mark, or dollar sign, and the third string contains an illegal character (i.e., %).

```
123      4BIT      SIZE%50
```

By default, the Macroassembler does not distinguish between uppercase and lowercase letters. For example, the Macroassembler interprets the symbol 'START' the same as the symbol 'start'.

Also by default, MASM recognizes the first five characters of a symbol. If you use longer symbols, MASM ignores the excess characters but does not return an error. Thus, the Macroassembler does not differentiate among the following three symbols (for symbol length equal to five):

```
FILES1     FILES2     FILES_TO_TAPE
```

The /8 switch lets you use the first eight characters of a symbol instead of the first five. Note that macro names must still be unique in the first five characters. Also, any symbol that is not a macro name must not have its first five characters the same as those of a macro name. MASM will not return a warning or error if this restriction is violated.

If you include the underscore character in a symbol that appears in a macro definition, precede that underscore character with another underscore (i.e., inside a macro, use A__B to represent the symbol A_B). "Macro Definition" in Chapter 5 provides more information on this subject.

Symbol Types

The Macroassembler recognizes three classes of symbols:

* Permanent symbols (pseudo-ops)

* Semipermanent symbols (instructions, macros, and system calls)

* User symbols

These symbol types vary according to the following criteria:

* Where the symbol definitions reside

* Whether you may redefine the symbols and, if so, how you redefine them

Symbol definitions may reside in one of three places: within the Macroassembler program, your source code, or the permanent symbol table (file MASM.PS or MASM16.PS (AOS/VS MASM16)).

The permanent symbol table is an external file that the Macroassembler uses to resolve symbols that are not defined in either the Macroassembler or your source code. For the purpose of the present discussion, symbols defined in the permanent symbol table are equivalent to those defined in your source code.

Table 2-6 summarizes the various symbol types. Refer to the following discussions for more information on the different symbols.

Table 2-6. The Four Classes of Symbols

Class	Symbols in Class	Where Defined	How to Redefine
Permanent	Pseudo-ops	Internally in Macroassembler software	Cannot be removed or redefined
Semipermanent	ECLIPSE assembly language instruction mnemonics, macro names, or system calls	In source code or permanent symbol table	Can be removed (via .XPNG) and redefined
User	Labels User variables System Parameters	In source code or permanent symbol table	Can be redefined at any time (without using .XPNG)

"Symbol Interpretation" in Chapter 3 describes how the Macroassembler translates the symbols in your source code. "Permanent Symbol Table" in Chapter 8 describes how to save symbol definitions for future use.

Permanent Symbols (Pseudo-Ops)

Permanent symbol definitions reside within the Macroassembler. You can neither delete nor redefine permanent symbols, as their name implies. Thus, the Macroassembler always recognizes them in your source code.

The permanent symbol set consists solely of pseudo-ops. Conversely, all pseudo-ops are permanent symbols. Pseudo-ops serve two purposes:

* They direct the assembly process.

* They represent numeric values of internal assembler variables.

Pseudo-ops that direct the assembly process are called assembler directives. Assembler directives set the location counter, reserve memory blocks, store ASCII text strings, control the program listing, and define macros, as well as perform other functions. When using an assembler directive in your program, you must adhere to the particular syntax for that symbol.

For example, the pseudo-op .TXT stores an ASCII text string in memory. When using .TXT, you must supply a single argument (i.e., the text string). If you do not supply an argument or you supply more than one argument, MASM returns an error. Chapter 7 specifies the syntax for each assembler directive pseudo-op.

In addition to directing the assembly process, pseudo-ops may represent internal assembler variables. When serving this purpose, the pseudo-op is called a value symbol.

For example, the symbol .PASS has the value of the current assembler pass number. During the first assembler pass, it has a value of 0; on the second pass, its value is 1. Another value symbol is the period (.). This single-character symbol has the value of the current location counter.

Depending on how you use them in your source program, certain pseudo-ops can either direct the assembly process or represent internal assembler variables. The Macroassembler uses the symbol's context in the source line to determine its intended use.

If the source line begins with the pseudo-op, the Macroassembler interprets that symbol as an assembler directive. If you pass the pseudo-op as an argument or use it as an operand in an expression, the Macroassembler interprets it as an internal variable (i.e., as a value symbol). MASM does not consider the label field when inspecting the pseudo-ops in a source line. To clarify these rules, consider the following examples.

You may use the pseudo-op .RDX either to specify a new radix for numeric input (as an assembler directive) or to represent the value of the current input radix (as a value symbol). In the line

```
.RDX            10 <nl>
```

the symbol .RDX begins the source line. Therefore, the Macroassembler interprets it as an assembler directive and sets the current input radix to 10 (decimal). However, in the source line

```
(.RDX) <nl>
```

the character "(" signifies the beginning of an expression. Thus, the Macroassembler interprets .RDX as a value symbol and generates a storage word with the numeric value of the current input radix (in this case, 10).

In the source line

```
SET_BASE:       .RDXO          .RDX <nl>
```

the Macroassembler interprets the first pseudo-op, .RDXO, as an assembler directive. The second pseudo-op, .RDX, is a value symbol because it appears as an argument. Pseudo-op .RDXO specifies a radix for numeric output from the assembler. Thus, the above source line sets the output radix equal to the current input radix. Note that the label SET_BASE does not affect the interpretation of the pseudo-ops. All pseudo-ops, both assembler directives and value symbols, begin with a period (.). Appendix B lists all pseudo-ops. Chapter 7 describes each symbol in detail.

Instruction Symbols

Instruction symbols are all the symbols in your ECLIPSE computer's assembly-language instruction set. The Macroassembler is supplied with files that have definitions for the standard ECLIPSE instructions. These files are not part of the Macroassembler program. Therefore, instruction symbols are semipermanent; they can be redefined within an assembly language program.

Each instruction symbol has a specific syntax associated with its use. As an example, consider the assembly language instruction LDA which loads a value from memory into an accumulator. When using LDA, you must supply an accumulator, a displacement value of 8 bits or less, and, optionally, an addressing index. In addition, LDA lets you specify indirect addressing by including the character @ in the source line's address field. The general syntax for LDA is

LDA□AC□<@>displacement<□index><nl>

When the Macroassembler encounters an instruction symbol, it scans the line to make sure the syntax is correct. If your source line does not conform to the syntax required by the instruction symbol, the Macroassembler returns an error. If your source line is syntactically correct, the Macroassembler sets bits in the appropriate fields of the instruction according to the values of the symbol and its arguments.

For example, when the Macroassembler encounters the instruction symbol LDA, it produces a one-word instruction as follows:

```
                     1 1 1 1 1 1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
|-----|---|-|---|---------------|
|0 0 1|A C|@|Idx| Displacement  |
|-----|---|-|---|---------------|
  LDA
 Opcode
```

When specifying an argument to an instruction symbol, be sure the value of that argument fits in the corresponding field of the instruction. If the field cannot contain the value, MASM returns an error. For example:

```
LDA 0,BUFF,1       ;Legal instruction.

LDA 4,BUFF,1       ;Illegal instruction--the first
                   ;argument (4) will not fit in the
                   ;instruction's 2-bit accumulator
                   ;field.
```

As a general rule, be sure that your argument values conform to the following equation:

$$\textbf{argument-value}\square\textbf{<=}\square(\textbf{2}^{\textbf{(field-width)}}\textbf{-1})$$

where:

argument-value is a value that you pass to an instruction symbol

field-width is the number of bits in the field corresponding to **argument-value**

To summarize the above discussion, the Macroassembler associates two formats with every instruction symbol:

A source format — that describes the correct use of the symbol in a source line. This format specifies the number of required and optional arguments you may pass to the instruction.

An assembly format — that describes the fields which receive the values of the instruction symbol and its arguments. Be sure the argument values you pass to an instruction symbol fit into the corresponding fields.

The <u>Programmer's Reference, ECLIPSE®-Line Computers</u> manual describes these formats for each 16-bit assembly language instruction.

Redefining Instruction Symbols

Instruction symbols, such as LDA, ESTA, and JMP, are semipermanent. Therefore, the Macroassembler lets you redefine their symbols. To do so, you can use the .XPNG pseudo-op or begin the assembly without a permanent symbol table. The .XPNG pseudo-op removes all of the Macroassembler's semipermanent (instruction, macro, and system call) symbol definitions. Then, you may assign a new value to any instruction symbol. For example:

```
.XPNG    ;Deletes all semipermanent symbol definitions
LDA=5    ;Defines LDA as a user symbol
```

The .XPNG statement removes LDA's definition; the Macroassembler will no longer interpret LDA as an instruction that loads a value into an accumulator. The second statement redefines LDA as a user symbol with the value 5 (we explain user symbols later in the chapter).

You can achieve the same effect if you assemble your program without a permanent symbol table. In this case, the Macroassembler will not recognize any instruction, macro name, or system call symbols. We describe this technique in Chapter 8.

You can also use a symbol table pseudo-op to define a new load-accumulator instruction with a different symbol (e.g., LOAD). We introduce these pseudo-ops in the "Defining Assembly Language Instructions" section of this chapter.

Macro Symbols

Every macro symbol represents a series of assembly language source statements. Whenever you wish to use those source statements in your program, simply place the macro symbol on a source line. The Macroassembler automatically substitutes the correct source statements for that symbol.

Use the .MACRO pseudo-op to associate a series of source statements with a symbol. You may choose any unique symbol name to represent those source statements. Be sure that your macro symbol conforms to the rules for symbol names presented earlier in the section "Symbol Names."

Macro symbols are semipermanent; you can redefine them within an assembly language program. We provide a complete discussion of macros in Chapter 5. Also, refer to the .MACRO pseudo-op description in Chapter 7 for more information.

In addition to your own macro symbols (those you define in your source), you may use the predefined macro symbols provided in the AOS software package. We call these predefined macros system calls; they perform frequently used operations and may greatly simplify assembly language programming. All system call names begin with a question mark (?). Their definitions reside in the permanent symbol table MASM.PS or MASM16.PS (see "Permanent Symbol Table" in Chapter 8). The AOS Programmer's Manual documents all system calls.

Redefining Macro Symbols

You can use the .XPNG pseudo-op to remove all macro symbol definitions (including system call definitions). After deleting a macro symbol's definition, you may assign that symbol a new value.

There is no way to reinstate a system call's definition during an assembly after you remove it. To reinstate other macros, you must list the appropriate source statements in a .MACRO declaration.

Refer to the .XPNG pseudo-op description in Chapter 7 for more information on redefining macro symbols.

User Symbols

User symbols are all the symbols that you define in your source program, except for macro symbols. Among their other functions, user symbols allow you to name locations symbolically, represent numeric values, and name external values. For example:

```
START:          A=3 <nl>
```

This statement defines both START and A as user symbols. START is a label whose value is the current address; A is a variable with the value 3.

User symbols do not have formats associated with their use. You can use them any place you can use integers.

User symbols may be either local or global. A local symbol has a value only for the duration of the single assembly in which it is defined. The value of a global symbol is known at link time; thus, you may use it in separately assembled modules. "Intermodule Communication" in Chapter 6 provides more information on this subject.

Reserved Symbols

When choosing user symbols for your program, be sure they conform to the general rules for symbol names presented earlier in this section. In addition, make sure that your user symbols do not conflict with any permanent, instruction, or macro symbols. Though MASM permits it, we recommend that you not use the question mark (?) as the first character in your user symbols; the system calls provided by the operating system all begin with a question mark. Do not use the .CALL, .KCALL, .RCALL, .RCHAIN, or .TARG symbols in your source code. These five symbols are reserved for the operating system's use.

Redefining User Symbols

Generally, you may change the value of a user symbol at any point in your program (without using .XPNG). The following sequence of source statements is perfectly legal:

```
START:          A=3
                A=A+A
                A=0
```

At the end of the above sequence, user symbol A has the value 0. You can use the .DUSR pseudo-op to define a user symbol that will not appear in the cross reference listing. For example, if you use the simple assignment statement A=3, the symbol A will appear in your program's cross reference listing. If you coded the statement .DUSR A=3, A would still equal 3, but A would not appear in the cross reference listing. You can assign a different value to a user symbol after it is defined with the .DUSR pseudo-op. However, this will produce a multiple definition (M) error if you have the /M switch in your command line (see "Command Line Switches" in Chapter 8).

Remember that, contrary to the general rule, you may not modify the value of a label. For example, the following code generates an error at assembly time:

```
        .NREL           0
LOC1:           .
                .
                .
        LOC1=LOC1+100               ;ERROR - DO NOT REDEFINE LABELS.
```

Expressions

An expression is

- a single user symbol, value symbol, or integer; or
- a series of user symbols, value symbols, and/or integers separated by operators.

The general format for an expression is

<sign>**operand**<operator□operand>...**break**

where:

sign — is one of the unary operators: (+ or -)

operand — may be a user symbol, a value symbol, an integer, or another expression

operator — is a Macroassembler operator (described in the next section); operands must both precede and follow every operator in your expression, except the unary operators

break — terminates the expression; the **break** character may be any delimiter or terminator (see "Terminators and Delimiters" earlier in this chapter)

According to this definition, the following strings are all legal expressions:

```
    START-1             6*3-5              A+3*B/C
```

You may not include any spaces within an expression, but you may use a space to terminate an expression. Thus, the Macroassembler does not view the string '3 + 5' as an expression because it contains spaces.

You may not use a symbol that has been defined with the .EXTN, .EXTD, or .COMM pseudo-op as part of an expression. For example, the following source code contains an illegal expression.

```
 .TITLE  MOD1
 .EXTN C
 C+1            ;Illegal expression.
.END
```

Operators

Table 2-7 lists all the operators that the Macroassembler recognizes.

Table 2-7. Operators

	Operator	Meaning
Arithmetic Operators	+	Addition (2+3) or unary plus (+3)
	-	Subtraction (5-4) or unary minus (-4)
	*	Multiplication
	/	Division
	B	Single-precision bit alignment (16 bits)
Logical Operators	&	Logical AND
	!	Logical OR
Relational Operators	==	Equal to
	<>	Not equal to
	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to

There are two different classes of operators:

- Binary operators

- Unary operators

Binary operators require two operands; one before and one after the operator. All the operators in Table 2-7 may function as binary operators. For example, the following expressions contain binary operators:

```
3+2      6*5      A<B       C!D
```

You may not place two binary operators in a row. The following are illegal expressions:

```
3*/4        8-*3    5**2    A*&B
```

The binary operators +, -, *, and / perform common mathematical operations; so, we do not describe their use in detail. However, the following sections of this manual provide information about the logical operators, the relational operators, and the bit alignment operators.

Unary operators require only one operand. The + and - characters can also function as unary operators. We provide more information in the next section.

Unary Operators

The Macroassembler can interpret the + and - characters as unary operators. The unary + and - operators simply indicate the sign of the following expression (positive or negative, respectively).

Unary operators require only one operand, which must appear immediately after the operator. Thus, a unary operator may either begin an expression or follow a binary operator within an expression. The following examples show the legal use of unary operators:

```
-5               +4               7*-3    6/+2
```

You may not place two unary operators together. For example, the following expressions are illegal:

```
--5              -+7              4*+-3
```

To legally apply two unary operators in a row, use parentheses to separate the operators. The following expressions illustrate this rule:

```
-(-5)          -(+7)          4*+(-3)
```

Logical Operators

The Macroassembler provides two logical binary operators: & and !. The operator & directs the Macroassembler to perform a logical AND operation; the operator ! represents the logical OR (inclusive) operation. To perform a logical operation, the Macroassembler must compare the bit patterns of both operands.

For a logical AND (&), the result in a given bit position is 1 only if both operands contain a 1 in that bit position. The following example shows how the Macroassembler evaluates the logical expression 6&4:

Bit representation of 6:	1 1 0
Bit representation of 4:	1 0 0
Result of logical AND (&):	1 0 0

Thus, the resulting value of the expression 6&4 is 4 (100 2).

For a logical OR operation (!), the result in a given bit position is 1 if either or both operands contain a 1 in that bit position. The following example shows the logical OR operation for the expression 6!4:

Bit representation of 6:	1 1 0
Bit representation of 4:	1 0 0
Result of logical OR (!):	1 1 0

The value of the expression 6!4 is 6 (110 2).

You must remember that you may not use relocatable operands in a logical expression; i.e., you may use only absolute operands. "Relocatability" in Chapter 3 provides more information about this subject.

Relational Operators

An expression containing a relational operator is a relational expression; that is, a relational expression contains one of the following:

```
<     <=     >     >=     <>     ==
```

A relational expression evaluates to either absolute zero (false) or absolute one (true). Absolute zero has a zero in every bit; absolute one has a one in the least significant bit (bit 15), and zeros in the rest of the bits. We refer to these values as "absolute" because they are not relocatable (Chapter 3 describes relocation).

The following examples show how the Macroassembler evaluates relational expressions (radix equals 8):

Assembled Value	**Expression**	**Comment**
000000	5==6	False
000001	3==3	True
000001	7>1	True
000000	55<=41	False
000001	7<>6	True

The octal numbers under the heading "Assembled Value" depict the 16-bit word that represents the expression's value. Remember, each data statement generates a single-precision integer (one word) by default.

The Bit Alignment Operator (B)

The Macroassembler recognizes B as a bit alignment operator. The B operator lets you right justify an integer on a bit boundary.

The general format for using the B bit alignment operator is

operand□bit-op□position

where:

operand is an integer, symbol, or expression whose value you wish to align

bit-op is the bit alignment operator (B)

position is an integer, symbol, or expression whose value indicates the bit position for aligning **operand**; MASM always interprets this value in decimal

When you use a bit alignment expression, the Macroassembler aligns the rightmost bit of **operand** at the bit position specified in **position**.

The bit alignment expression must not contain any spaces.

Use the B operator to generate single-precision (16-bit) integers. When using B, the value of **position** must be in the range

$$\mathbf{0\square{<=}\square position\square{<=}\square 15_{10}}$$

The result of a B bit alignment expression equals

$$\mathbf{operand_r*2(^{(15.\square-\square position)})}$$

where:

operand is the integer, symbol, or expression you want to align

r is the current input radix

position indicates the bit position for aligning **operand;** the Macroassembler evaluates **position** in decimal

The following examples illustrate the use of B (radix equals 8):

Assembled Value	**Expression**	**Comment**
000004	1B13	Aligns the value 1 in bit position 13_{10} i.e., $0\text{-}000\text{-}000\text{-}000\text{-}000\text{-}100_2$).
000012	5B14	Aligns the rightmost bit of 5_8 (101_2) in bit position 14_{10} (i.e., $0\text{-}000\text{-}000\text{-}000\text{-}001\text{-}010_2$).
000000	3B17	ERROR: the value 17 is outside the legal range for the B operator (there is no bit 17_{10} in a single-precision integer).
070000	27B3	Aligns the rightmost bit of 27_8 (010 111_2) in bit position 3. MASM truncates the portion of 27 that does not fit into the word.

If you use an input radix of 12 or greater, the Macroassembler interprets the character B as a digit instead of an operator. To avoid ambiguity, place the **operand** value inside parentheses; for example:

```
.RDX    16                  ;Input radix equals 16.
31B4                        ;The Macroassembler interprets
                            ;B as a hexadecimal digit.
(31)B4                      ;The Macroassembler recognizes
                            ;B as the bit alignment operator.
```

Priority of Operators

You may use more than one operator in an expression. The Macroassembler evaluates operators according to their priority levels. It resolves high priority operators first and low priority operators last. Table 2-8 lists the priority levels of all operators.

Table 2-8. Operator Priority Levels

Operators	Priority Level
B	3 (highest priority)
+ - * / & !	2
< <= > >= == <>	1 (lowest priority)

If an expression contains operators of equal priority, the assembler evaluates them from left to right.

The following examples show how MASM uses operator priority to evaluate expressions; the radix equals 8:

Assembled Value	Expression	Comment
000001	3*2==6	MASM evaluates * first, then ==. The relationship is true.
000000	4<6-3	MASM evaluates - first, then <. The relationship is false.
000005	4/2+3	/ and + are equal in priority so the assembler evaluates them from left to right (first /, then +).
000011	3*2-1+4	All operators are of equal priority so MASM evaluates them from left to right (first *, then -, then +).

000006	3!1*2	Both operators are of equal priority so MASM evaluates them from left to right (3!1=3, then 3*2=6).
000001	4&2<>3!1	MASM evaluates & first, followed by !, and lastly, <>. The resulting relationship is true.
030000	2*3B4	B has higher priority than *, so the assembler evaluates 3B4 first. It then multiplies the result by 2.

You may change the order in which the Macroassembler evaluates operators by including parentheses in your expression. The Macroassembler always evaluates an expression in parentheses first. Within a set of parentheses, MASM evaluates operators according to the priority sequence presented above. If you nest one set of parentheses inside another set, MASM evaluates the innermost expression first.

The following examples show the use of parentheses in expressions; the radix equals 8:

Assembled Value	**Expression**	**Comment**
000006	2*(4-1)	MASM performs operations in the following order: (4-1)=3. 2*(3)=6.
000002	1+(6/2)/2	Order of operations: (6/2)=3. 1+(3)=4. (4)/2=2.
000004	(3*2)-(4/2)	Order of operations: (3*2)=6. (4/2)=2. (6)-(2)=4.
000001	(5<=5)+(6==2)	Order of operations: (5<=5)=1. (6==2)=0. (1)+(0)=1.
000006	3*((3+1)/2)	Order of operations: (3+1)=4. ((4)/2)=2. 3*((2))=6.

Absolute Versus Relocatable Expressions

In the previous discussion, all the expressions contain integers or symbols defined by integers. We refer to these operands as absolute because their values are explicitly stated in the source module. Cther operands are relocatable; that is, their values relate to and are dependent upon other words in your module.

If you use relocatable operands in your expressions, you must follow certain rules, or the assembler will return an error. In Chapter 3, we discuss relocatable values and how to use them in your expressions.

Special Atoms

Three atoms do not fall into any of the previously discussed categories. They are @, #, and **.

At Sign (@)

You may include the at sign (@) in certain memory reference instructions (MRIs) or in any data statement.

When the Macroassembler encounters the @ character, it sets the assembled instruction's indirect addressing bit to 1. In one-word MRI instructions, bit 5 is the indirect addressing bit. In extended MRI instructions, an @ sign sets bit 0 of the instruction's second word. Refer to the Programmer's Reference, ECLIPSE®-Line Computers manual to determine which MRI instructions allow indirect addressing. The @ sign can appear anywhere in memory reference instructions. However, it is good practice to place it at the beginning of an instruction's displacement field.

In data statements, the first bit (bit 0) is the indirect addressing bit. You should place the @ sign immediately before the data in your source code.

The following examples show the use of the indirect addressing sign (@):

Assembled Value	Source	Statement
024020	**LDA**	**1,20**
026020	**LDA**	**1,@20**
142070 000020	**ESTA**	**0,20,0**
142070 100020	**ESTA**	**0,@20,0**
000025	**25**	
100025	**@25**	

Number Sign (#)

You may place the number sign (#) in any one-word ALC instruction. You may not use the number sign with multiword ALC instructions. When the Macroassembler encounters the # character, it places a 1 in that instruction's no-load bit (bit 12). At execution time, the arithmetic and logic unit (ALU) does not load the results of the operation into the destination accumulator (ACD).

The following examples illustrate the use of the no-load sign (#):

Assembled Value	Source	Statement
112405	**SUB**	**0,2,SNR**
112415	**SUB#**	**0,2,SNR**
133102	**ADDL**	**1,2,SZC**
133112	**ADDL#**	**1,2,SZC**

If you use the # character in a source line, you must also have a skip mnemonic in the line. If you do not, MASM returns an error, because those bit combinations represent other instructions in the ECLIPSE instruction set.

Refer to the Programmer's Reference, ECLIPSE®-Line Computers manual for a list of the skip mnemonics and for information about one-word ALC instructions.

Asterisks (**)

Two consecutive asterisks (**) in a source line direct the Macroassembler not to list that line. We suggest that you put the asterisks at the beginning of a source line. The asterisks do not affect the object module, only the assembly listing.

Refer to "Assembly Listing" in Chapter 4 for more information on the two asterisks (**) atom.

Statement Format

In general, all source statements in your module should use the following format:

```
        label:      statement-body      ;comment <nl>
```

Each nonblank line in your source module must contain a value for at least one of these three fields.

The Macroassembler is a free-form assembler. That is, the assembler is not column sensitive. Instead, it distinguishes between fields by searching for delimiters. For example, the label field delimiter is the colon (:). Table 2-9 lists the characters that delimit each field in your source line.

Table 2-9. Statement Field Delimiters

Field	Delimiter
label	Colon (:)
statement-body	Semicolon (;) or statement terminator (i.e., carriage return, form feed, or NEW LINE).
comment	Begins with a semicolon (;) and ends with a statement terminator.

You may include extra spaces and tabs between the statement fields in your source line without affecting the assembler's interpretation of that line. Thus, all of the following lines are equivalent:

```
label:statement-body;comment<nl>


        label:          statement-body          ;comment   <nl>


label:           statement-body                  ;comment <nl>
```

A common programming practice is to divide each source line into columns using the tab settings. Thus, **label** starts in the leftmost column; **statement-body** starts at the first tab stop, etc.

The maximum allowable length for a source statement is 132 characters. The Macroassembler truncates lines that are too long, and returns an error.

```
** label:              statement-body              ;comment <nl>
```

Refer to "Assembly Listing" in Chapter 4 for a discussion of the no-listing indicator (**).

The following sections of the manual discuss the three source statement fields: **label, statement-body,** and **comment.**

Labels

A label symbolically names a memory location. By using labels, you can refer to locations without regard for numeric addresses.

Any source statement can have a label. It must appear at the beginning of the source line and must be followed by a colon (:). All labels must conform to the rules for symbol names (see "Symbol Names" earlier in this chapter).

The following source lines have examples of how to use labels.

```
BEGIN:          ELDA        0,SEVEN
JUMP:           JMP         @17
SEVEN:          7
```

Like other symbols, a label has a value. The value of a label equals the value of the current location counter. MASM computes the label value prior to processing the rest of the source line. Thus, a label usually equals the address of the next storage location that the assembler creates. (Be sure to read about the exception case at the end of this section.) You may not alter the value of a label at any point in your program.

According to these rules, the label BEGIN in the first line of the previous example receives the address of the assembled ELDA instruction as its value. Location SEVEN contains the value 7.

Since some source lines do not generate storage words, a label is not necessarily associated with the source statement it appears in. For example,

```
START:          .TITLE         MOD1
                LDA            0,1
```

Here, the first statement assigns the title MOD1 to the source module, and does not generate a storage word. Therefore, the label START receives as a value the address of the next location assembled; in this case, the address of the LDA instruction.

Similarly, a label may appear alone on a line. In this case, its value equals the address of the next storage location assembled.

```
LABEL1:
                LDA            0,1
```

In this example, the value of LABEL1 equals the address of the assembled LDA instruction.

```
LABEL1:
LABEL2:         LDA             0,1
```

Here, both LABEL1 and LABEL2 equal the address of the LDA instruction.

You may place more than one label on a source line; all labels will receive the same value. For example:

```
LOOP1: LOOP2: LOOP3:                ADD         0,1
```

LOOP1, LOOP2, and LOOP3 all equal the memory address of the assembled ADD instruction.

In the previous examples, all labels receive the address of the next location MASM created. However, this is not the case if your label appears on a source statement that alters the value of the location counter. Since it computes a label's value before evaluating the source line, MASM may never actually create the location it assigns to the label. Consider the following example:

```
        .LOC        100     ;Set the location counter to 100.
A:      .LOC        .+50    ;Increase the location counter by 50.
B:      ELDA        0,1
          .
          .
          .
```

The first statement directs MASM to start assigning addresses at absolute location 100. When MASM encounters label A on the next statement, it immediately assigns that label the value 100. Before MASM actually creates location 100, however, the second .LOC statement changes the value of the location counter to 150. Thus, label B's value (150) is the address of the ELDA instruction, but label A's value (100) does not identify an allocated location. References to address A produce unpredictable results.

Any of the following pseudo-ops may change the value of the location counter:

```
        .GLOC
        .LOC
        .NREL
        .ZREL
```

In general, you should not place labels on these pseudo-op statements. In fact, except for .BLK and .TXT, you need not place a label on any pseudo-op statement.

Statement Body

The statement-body field of a source line may contain one of the following:

* Assembly language instruction

* Macro or system call

* Pseudo-op directive

* Assignment

* Data

We discuss these five types of source statements later in this chapter (see "Statement Types").

Comments

You should include comments in your programs. This will help you develop, test, and document your programs. The assembler does not interpret comments. Therefore, comments do not affect the generation of the object module.

Precede all comments with a semicolon (;). When the assembler encounters a semicolon, it ignores all subsequent characters up to the statement terminator.

The following source statements show the use of comments:

```
    LDA     0,A      ;Get counter's initial value
    LDA     1,B      ;Get the value that increments the counter
    ADD     0,1      ;Produce the counter's new value


A:  63               ;Counter's initial value
B:  44               ;Initial value of counter's increment


                ;Also note that comments can appear alone on a line.
```

Statement Types

As we mentioned previously, there are five different types of source statements:

* Assembly language instructions

* Macros and system calls

* Pseudo-op directives

* Assignments

* Data

Each statement type must conform to the general statement format that we presented in the previous section. Thus, the general statement format is now

```
              ( assembly language instruction )
              | macro or system call          |
     label:  <  pseudo-op directive            >  ;comment <nl>
              | assignment                    |
              ( data                          )
```

In addition to the general statement format, each of the five statement types also has a syntax specific to that type.

The next sections describe the five statement types and the syntax for each.

Assembly Language Instructions

Assembly language instructions perform specific operations at execution time. There are three categories of assembly language instructions:

* Arithmetic and logic (ALC)

* Memory reference (MRI)

* Input/output (I/O)

Arithmetic and logic (ALC) instructions let you manipulate data in the CPU. That is, the ALC instructions perform operations on data residing in the accumulators (e.g., add, subtract, complement, logical AND).

Memory reference instructions (MRIs) let you perform the following operations:

* Modify the program counter (PC)

* Modify an operand in memory

* Transfer data from memory to an accumulator

* Transfer data from an accumulator to memory

I/O instructions let you communicate with peripheral devices. In particular, these instructions

- start and stop peripheral devices,

- transfer data from a device to an accumulator in the central processing unit (CPU),

- transfer data from an accumulator to a device, or

- test the status of a device.

Note that you *may not* use I/O instructions in your module if that program will run under an operating system. Instead, you must use I/O system calls (see "Macros and System Calls" below).

The syntax for using an assembly language instruction in your source module is

instr<□*arg*>...

where:

instr is an assembly language instruction mnemonic; all such mnemonics are instruction symbols. Refer to "Instruction Symbols" earlier in this chapter for more information about the properties of this symbol type.

arg is an argument to the assembly language instruction. Not all instructions require arguments; some require many.

The *Programmer's Reference, ECLIPSE®-Line Computers* manual describes the 16-bit ECLIPSE computers' assembly language instructions. It also specifies what arguments you must supply to each instruction. Examples of four typical ECLIPSE assembly language instructions are

```
    LDA       0,-2,1     ;Loads accumulator 0
    ADD       2,3        ;Adds two values
    JMP       START      ;Jumps to the location named by START
    HALT                 ;Halts the CPU
```

The Macroassembler lets you define your own assembly language instructions. We introduce the pseudo-ops that let you redefine instructions at the end of this chapter.

Macros and System Calls

A *macro* is a series of assembly language source statements that you assign a name. Whenever you wish to place that section of source code in your source module, you simply enter the macro name; the assembler substitutes the corresponding code.

The syntax of a macro call is

macro-name<□arg>...

where:

macro-name is the name you assign to a series of assembly language source statements

arg is an argument to the macro

In Chapter 5, we explain how to create and use macros. Chapter 3 describes how the Macroassembler processes macros.

Included in the AOS software package is a set of predefined macros. Some of these system-defined macros perform frequently used operations. We refer to these macros as system calls. The advantage to using system calls is that you do not have to code the macros yourself; simply issue the system call.

The syntax for issuing a system call is

?system-call<□arg>...
exception-return
normal-return

where:

?system-call is the name of a system call. All system calls begin with a question mark. Every system call mnemonic is a macro symbol (see "Macro Symbols" in this chapter).

arg is an argument to the system call.

exception-return is a one-word source statement that gains control if the system call causes an error or exceptional return.

normal-return is a source statement that gains control if the system call does not cause an error or exceptional return.

The following are examples of system call statements:

```
?OPEN    TEST  ;Open file defined by parameter packet TEST
?WRITE   TEST  ;Write record to file defined by packet TEST
?CLOSE   TEST  ;Close file defined by parameter packet TEST
?RETURN
```

Note that each of these examples shows only the system call itself, not the exceptional and normal-return statements.

The AOS Programmer's Manual describes all system calls in detail. Chapter 3 in this manual explains how the Macroassembler processes them.

Pseudo-Ops

A pseudo-op, also called an assembler directive, directs the operation of the Macroassembler. It is called a "pseudo-op" because your program never executes it; rather, the assembler executes it.

In addition to performing other functions, pseudo-ops

- tell the assembler where in memory your source code is to reside,
- allow separately assembled source modules to communicate with each other, and
- define macros
- define instructions
- control the listing output format
- allow for conditional assembly of selected statements

The syntax for a pseudo-op source statement is

.pseudo-op<□arg>...

where:

.pseudo-op is the name of a pseudo-op. All pseudo-ops begin with a period (.) and every pseudo-op mnemonic is a permanent symbol. Refer to "Permanent Symbols" earlier in this chapter for more information about this class of symbols.

arg is an argument to the pseudo-op.

The following are examples of pseudo-op source statements:

```
.ZREL
.NREL        1
.TITLE       MOD1
.ENT         GLOBE
.RDX         8
```

Note that you may use certain pseudo-op symbols as values in other source statements. For example,

```
    X=.RDX
```

assigns the value of the current input radix to the variable X. This statement is not a pseudo-op directive but rather an assignment (see "Assignments" below).

When using a pseudo-op symbol in this fashion, we refer to it as a value symbol. We discuss value symbols under "Permanent Symbols" earlier in this chapter.

Chapter 6 discusses the different types of pseudo-ops. Chapter 7 describes the pseudo-ops individually and specifies what arguments you must supply to each.

Assignments

An assignment statement assigns a single-precision (16-bit) integer value to a symbolic name. After associating a value with a symbol, you may use the symbol any time you wish to indicate the value.

The syntax of an assignment statement is

$$\textbf{user-symbol=} \quad \left\{ \begin{array}{l} \textbf{integer} \\ \textbf{symbol} \\ \textbf{expression} \\ \textbf{instruction} \end{array} \right\}$$

where:

user-symbol is a user symbol conforming to the rules for symbols (given earlier in this chapter); the Macroassembler assigns **user-symbol** the value on the right side of the = character

integer is any integer value; you may not place a floating-point number on the right side of an assignment statement

symbol is any user symbol or value symbol

expression is any legal expression

instruction is any legal ECLIPSE 16-bit assembly language instruction

Examples of assignment statements are

```
        A=322
        B=10*3
        C=(A/2)+B
        D=C
        E=.RDX
        F=(.PASS+10)
        G=ADD          0,1
        H=LDA          0,1
```

If you place an instruction on the right side of an assignment statement, MASM computes the assembled value of that instruction and assigns it to the variable on the left side of the statement. The Macroassembler issues an error if an instruction's assembled value is greater than 16 bits. Table 2-10 shows how MASM assigns instruction values to user symbols.

Table 2-10. Instructions in Assignment Statements

Assignment	16-Bit Value Assigned by MASM
M=STA 1,0	044000
N=LDA 0,1	020001
O=JMP @100	002100

You may not include instructions within expressions. That is, the following assignments are illegal and generate errors:

```
        P=(ADD  0,0)+5
        Q=(ELDA  0,0,0)-(LDA   0,1)
```

You may not include the .EXTD, .EXTN, or .COMM pseudo-ops in expressions. Nor can an expression include a symbol that has been defined by one of these pseudo-ops.

Data

A data statement is one of the simplest assembly language statements you will use in your program. It consists of a single number, symbol, or expression. When the assembler encounters a data statement, it simply evaluates the number, symbol, or expression and stores the value in memory.

Examples of data statements are

```
0
322
10255
32*5
5.3E4
A/2
SIX
```

The Macroassembler generates a one-word (16-bit) storage area for each data statement, by default. To store data in two 16-bit words, you must use the D indicator (see "Double-Precision Integers" in this chapter for more information).

You may include the special character @ in a data statement. The assembler then places a 1 in bit 0 (the indirect addressing bit) of the storage word. The @ sign should immediately precede the data on the source line; for example:

```
@113
@1032
```

Refer to "At Sign (@)" earlier in this chapter for more information.

Defining Assembly Language Instructions

The Macroassembler has a set of pseudo-ops that let you:

* change a standard instruction's mnemonic;

* add your own instructions to the standard instruction set; or

* create your own instruction set.

New instruction mnemonics can be saved and used for other assemblies. For example, you could use the .DMR pseudo-op to change the JMP instruction's mnemonic to JUMP. You could also use the .DIAC pseudo-op to define a new instruction symbol (called WAIT) that "requires an accumulator." When you use the WAIT symbol in a program, the Macroassembler scans for an expression following the symbol. If no expression is found, a format error is generated on the source line. If an expression is present, its value determines the value of the instruction's accumulator field.

Instruction symbols are semipermanent; they can be saved and used, without redefinition, for all subsequent assemblies. You can add to the standard ECLIPSE instruction definitions, or you can eliminate the standard instructions and define your own.

Table 2-11 lists the the kinds of ECLIPSE assembly language instructions and the pseudo-ops that you can use to define new instructions.

Table 2-11. Instruction Definition Pseudo-Ops

Instruction Type	Example	Defining Pseudo-Op
Arithmetic and logical (ALC)	ADD	.DALC
Extended ALC, two accumulators, no skip	IOR	.DISD
Extended ALC, two accumulators, skip	SGT	.DISS
Memory reference (MRI)	JMP	.DMR
Extended memory reference	EJMP	.DEMR
Memory reference with accumulator	LDA	.DMRA
Extended memory reference with accumulator	ELDA	.DERA
Commercial memory reference	ELDB	.DCMR
Count and accumulator	ADI	.DICD
Extended immediate	ADDI	.DIMM
I/O without accumulator	NIO	.DIO
I/O with two required fields	DIA	.DIOA
Instruction requiring an accumulator	XCT	.DIAC
Extended with one argument field	SAVE	.DEUR
Extended operation	XOP	.DXOP
Floating point load/store	FLDS	.DFLM
Floating point load/store status, no accumulator	FLST	.DFLS
Define a user symbol as semipermanent without argument fields	-----	.DUSR

The following sample of source code deletes all of the standard instruction symbols and defines a LOAD, STORE, and JUMP instruction. These instructions correspond to the standard LDA, STA, and JMP instructions.

```
    .XPNG                       ;Deletes all semipermanent symbols

    .DMRA  LOAD=020000          ;Defines LOAD as a MRI instruction
                                ;requiring an accumulator (same as
                                ;standard LDA instruction)

    .DMRA  STORE=040000         ;Defines STORE as a MRI instruction
                                ;requiring an accumulator (same as
                                ;standard STA instruction)

    .DMR   JUMP=000000          ;Defines JUMP as a MRI
                                ;(same as standard JMP instruction)
```

Refer to the symbol table pseudo-op descriptions in Chapter 7 for more information about redefining instruction symbols. The Programmer's Reference, ECLIPSE®-Line Computers manual describes the formats for different kinds of ECLIPSE instructions.

End of Chapter

Chapter 3
The Assembly Process

As we discussed in Chapter 2, your assembly language source module is a series of ASCII characters grouped into source statements. The Macroassembler interprets those statements and produces a binary representation of your source module. The resulting binary module is called an object module.

MASM is a two-pass assembler; that is, it scans your source code twice to produce an object module. During these two passes through your source code, the Macroassembler performs four major functions:

- Interprets symbols
- Checks source statement syntax
- Expands macros and system calls
- Resolves memory locations

In the following sections, we describe how the Macroassembler performs these functions.

Note that the above list of assembler functions is by no means complete. For example, the Macroassembler can produce a variety of listings, conditionally assemble code, and generate a symbol table for future use. Though these are important capabilities of the Macroassembler, they are not its major functions. Thus, we describe these additional features in other chapters of the manual.

Symbol Interpretation

Chapter 2 describes the kinds of symbols (i.e., pseudo-op, instruction, macro, and user) that may appear in your source module. As the Macroassembler processes your source module, it translates all symbols into their machine-readable binary equivalents.

The Macroassembler uses definitions from its internal database and two external tables to resolve the symbols in your source code. Pseudo-op definitions are contained in the Macroassembler's internal database. The two tables are the permanent symbol table, and the temporary symbol table. They contain instruction, macro, and user symbol definitions. The permanent symbol table stores definitions that you want to use for several assemblies (e.g. instruction or macro definitions). The Macroassembler creates a temporary symbol table for each assembly.

During the assembly process, the Macroassembler makes sure that all symbols are defined in the database or temporary symbol table (adding definitions from your source, if necessary). On its second pass through your program, the assembler uses these definitions to produce binary code for the symbols in your source module. The following discussion shows how the Macroassembler interprets symbols. Refer to Figure 3-1 as you read this section.

Symbol Tables

The Macroassembler defines all permanent symbols (pseudo-ops) in its internal database. Thus, the Macroassembler always recognizes any pseudo-ops in your source module. You cannot change the definitions in this database.

When you issue a MASM command line, the Macroassembler automatically creates a temporary symbol table. It copies the contents of the permanent symbol table into the temporary table. In most cases, you will use the permanent symbol table we supply with the AOS software package. This table (MASM.PS or MASM16.PS (AOS/VS MASM16)) contains definitions for the standard ECLIPSE assembly language instructions and AOS system calls (e.g., ?OPEN, ?READ) and system parameters. Chapter 8 explains how to build your own permanent symbol table.

When the Macroassembler encounters a symbol in your program, it first checks to make sure the symbol is valid (i.e., conforms to the rules for legal symbols presented in Chapter 2). MASM returns an error if a symbol is not legal.

If the symbol is valid, the Macroassembler looks for its definition within its database. If it's there (i.e., if the symbol is a pseudo-op), the assembler need not check the temporary symbol table for that symbol's definition.

If the symbol is not a pseudo-op, the Macroassembler tries to resolve it by checking the temporary symbol table. The temporary symbol table starts as a copy of the permanent symbol table (usually MASM.PS or MASM16.PS (AOS/VS MASM16)).

If the Macroassembler cannot find the symbol's definition in its database or temporary symbol table, it checks to see if the current source statement defines the symbol. If so, the Macroassembler copies this definition into the temporary symbol table. If the table already contains the symbol, MASM updates its definition according to the information in the source line. After modifying the temporary symbol table, the Macroassembler moves to the next symbol in your source and repeats the symbol resolution process.

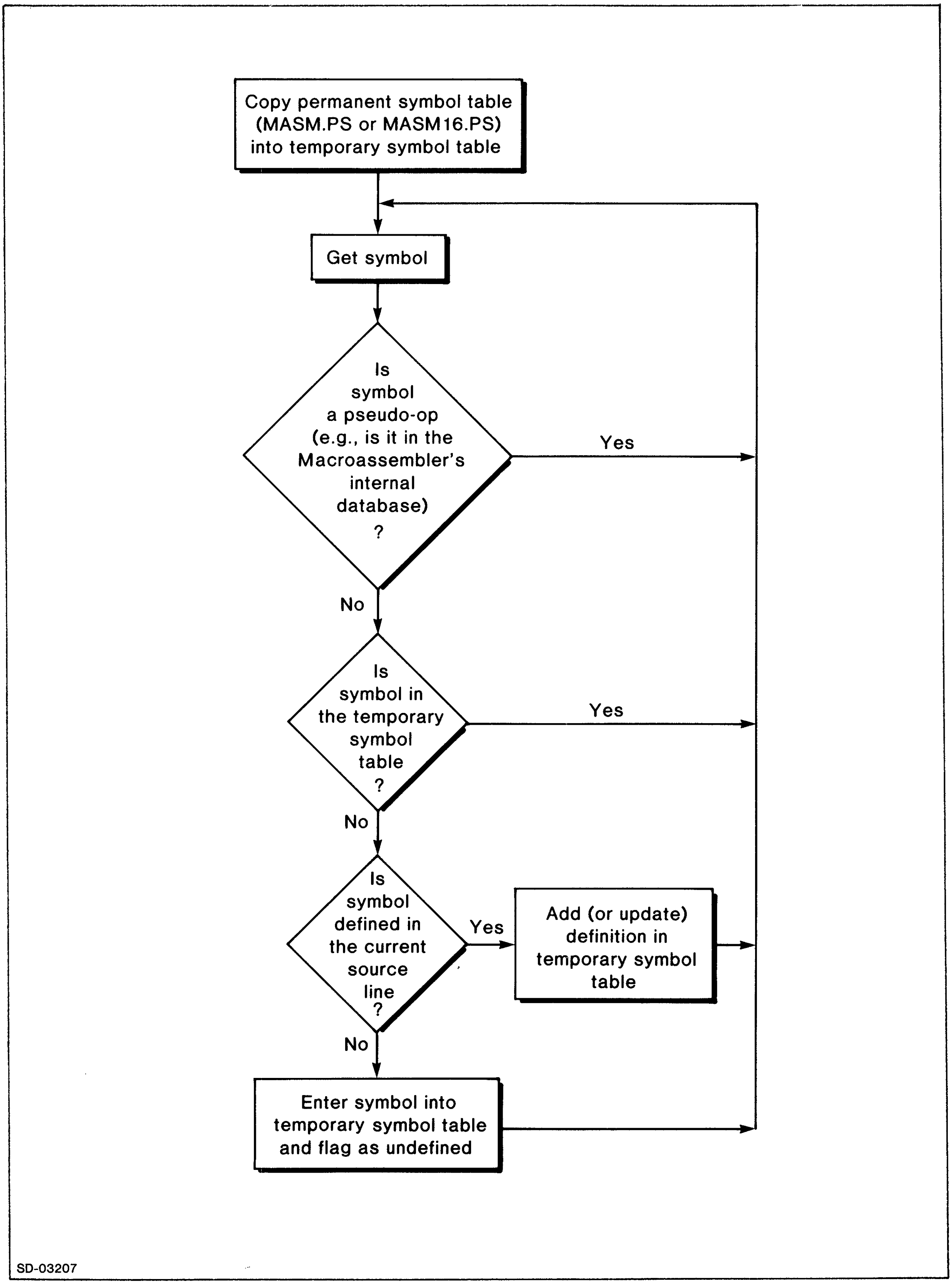

Copy permanent symbol table (MASM.PS or MASM16.PS) into temporary symbol table
Get symbol
Is symbol a pseudo-op (e.g., is it in the Macroassembler's internal database) ?
Yes
No
Is symbol in the temporary symbol table ?
Yes
No
Is symbol defined in the current source line ?
Yes
Add (or update) definition in temporary symbol table
No
Enter symbol into temporary symbol table and flag as undefined
SD-03207

Figure 3-1. Resolving Symbols

If the Macroassembler does not find a symbol's definition in the database, temporary symbol table, or current source statement, it enters that symbol in the temporary table and flags it as undefined. MASM then moves on to the next symbol in your source.

Note that the Macroassembler may not find a symbol's definition the first time that symbol appears in your source. However, subsequent source statements may define it. For example, the first part of your source program may reference a label that is defined towards the end of that program.

```
     .
     .
     LDA 0,A
     ADD 0,1
     .
     .
A:   3
```

In this example, the first time MASM encounters the symbol A, it searches its database, the temporary symbol table, and the current source statement for A's definition. Not finding it, MASM enters A in the temporary symbol table and flags it as undefined. Later, when it discovers that A is a label, MASM will place the appropriate value in the table.

During pass two, the Macroassembler uses its database and the temporary symbol table to translate the symbols in your module into binary code. When it finds the symbol's definition, MASM substitutes the corresponding code in the object file.

If, during pass two, the Macroassembler encounters a symbol that is not defined in its database or temporary symbol table, MASM returns an error when it tries to substitute a value for that symbol. If you indicate that a separately assembled module defines a symbol (i.e., with an intermodule communication pseudo-op), the Macroassembler does not return an error. (Chapter 6 contains more on intermodule communication.)

Syntax Checking

As we explained in Chapter 2, all source statements and their component parts must conform to specific syntaxes. The following discussion describes how the Macroassembler checks your source module for syntax errors. This presentation is a general overview and does not list all the syntax rules; we provide those rules in appropriate places throughout the manual.

When the Macroassembler scans your source module, it must determine whether a given string of ASCII characters is a valid assembly language statement. Thus, the Macroassembler must first divide your module into distinct source statements by searching for statement terminators.

The Macroassembler then divides each statement into a series of atoms or syntactic units (see Chapter 2). During this process, MASM rejects any character that is not in the legal Macroassembler character set.

After isolating the atoms in each line, MASM determines whether those atoms form a legal source statement. If they do, MASM processes the statement according to its type (see "Statement Types" in Chapter 2). If the atom sequence is not a legal statement, MASM returns an error.

The remainder of this section explains how MASM evaluates statements. Figure 3-2 provides an overview of the operations involved and will help you understand the following discussion.

MASM starts processing each source line by evaluating the first atom. If the atom is not a symbol, MASM assumes that the line is a data statement. The Macroassembler knows that a data statement consists of a single expression followed by an optional comment string. If the atoms on the current source line do not conform to this format, MASM returns an error. The following statements generate errors because they do not conform to the format for data statements:

```
12+3            17              20
100:
5+10*2          LDA             0,1
```

If a source line conforms to the data statement format, MASM stores the data value in memory and moves on to the next statement.

If the first atom on a source line is a symbol, MASM determines whether a colon (:) follows it. If so, MASM assumes the symbol is a label and makes sure it is legal (see "Labels" in Chapter 2). At this point, the following statements would cause errors because pseudo-ops and instruction mnemonics may not appear as labels:

```
.RDX:           10
ADD:            77
```

If the symbol is an acceptable label, MASM enters it in the temporary table along with the appropriate value.

Labels may appear on any source line and do not place any restrictions on a statement's format and content. Thus, after MASM processes the label, it treats the atom following the colon as if it were the first atom on the source line.

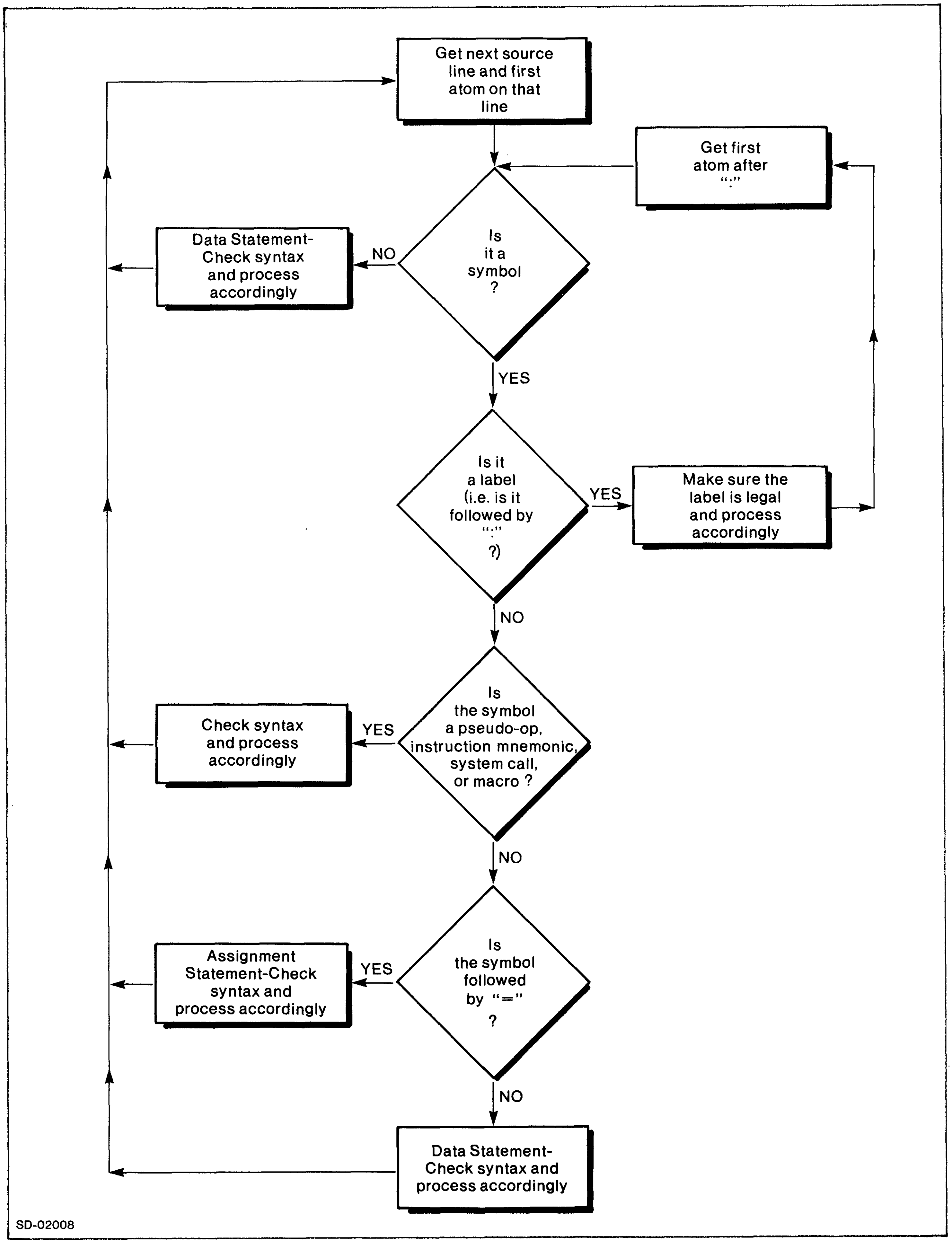

Get next source line and first atom on that line
Get first atom after ":"
Is it a symbol ?
NO
Data Statement-Check syntax and process accordingly
YES
Is it a label (i.e. is it followed by ":" ?)
YES
Make sure the label is legal and process accordingly
NO
Is the symbol a pseudo-op, instruction mnemonic, system call, or macro ?
YES
Check syntax and process accordingly
NO
Is the symbol followed by "=" ?
YES
Assignment Statement-Check syntax and process accordingly
NO
Data Statement-Check syntax and process accordingly
SD-02008

Figure 3-2. Processing Source Statements

If the first atom on the source line is a symbol and is not followed by a colon, MASM checks its database and temporary symbol table to see if it is a pseudo-op, an instruction mnemonic, a system call, or a macro. If so, MASM makes sure the rest of the atoms on that source line conform to the syntax implied by the first symbol. That is, MASM ensures that the number of atoms and their values conform to the rules associated with the first symbol's use. At this point, MASM would return errors for the following statements (we put the reason for the errors in the comment fields):

```
LDA              0                ;Not enough arguments.
.PUSH                             ;Requires an argument.
ELDA             15,LOC           ;Illegal AC value.
.PASS=10                          ;Illegal syntax format.
LDA              0,0,0,0          ;Too many arguments.
.RDX             21               ;Illegal argument value.
```

If the first symbol on a source line is a pseudo-op, instruction mnemonic, system call, or macro and if the other atoms on the line conform to that symbol's use, MASM performs one of the following operations:

Pseudo-op	Interprets and performs the appropriate action
Instruction	Assembles and stores it in memory
Macro or System Call	Expands it (see "Processing Macros and System Calls" later in this chapter)

If the first atom on the source line is a symbol but is not a label, pseudo-op, instruction, system call, or macro, MASM determines whether the equal sign (=) follows it. If so, MASM assumes that the statement is an assignment. Again, MASM checks to make sure the rest of the atoms in the line conform to the syntax rules for an assignment statement. The following assignment statements would generate syntax errors (see "Assignments" in Chapter 2):

```
A=10             20               30
B=
C=100=200
```

If the assignment statement is legal, MASM evaluates the expression on the right side of the statement and stores its value in the temporary symbol table with the symbol on the left side of the statement.

If the first atom in a source statement is a symbol but is not a label, pseudo-op, instruction, system call, or macro, and does not precede an equals character (i.e., is not an assignment), then

MASM assumes that the statement is a data entry. In this case, MASM makes sure the rest of the line conforms to the data statement syntax.

Processing Macros and System Calls

Chapter 2 describes macros and system calls. A macro is a named section of source code. Whenever you wish to insert that code in your module, simply specify the macro name. System calls are predefined macros that the AOS software package provides for your use. You must use system calls to perform I/O operations when you run your program under AOS.

The following discussion explains how the Macroassembler processes macros and system calls. Refer to Chapter 5 for information about using macros in your source module. The AOS Programmer's Manual explains system calls in detail.

Processing Macro Definitions

When you define a macro, you associate a symbol with a series of assembly language statements: the macro definition string. When it encounters a macro definition during pass one, MASM places the macro name in the temporary symbol table. MASM then copies the macro definition string into a different part of the temporary symbol table and places a pointer to that string with the macro name.

The Macroassembler does not check the syntax of the macro definition string at this time. It simply copies the definition directly into the temporary symbol table.

"Macro Definition" in Chapter 5 explains how to define a macro in your source module.

Expanding Macros and System Calls

The Macroassembler does not distinguish between system calls and calls to macros that you define in your source. Therefore, the following discussion applies to both forms of macros.

You should define a macro before you call it. When MASM encounters a macro definition, it copies the macro's name and definition into the temporary symbol table.

When you call a macro, MASM looks for its name in the temporary symbol table. If MASM finds the macro name in the table, it expands the macro. That is, the Macroassembler processes the macro definition string that resides in the temporary symbol table as if it was in your source module. During this operation, MASM checks the macro definition string for syntax errors. It also substitutes any arguments you supply in the macro call for formal (dummy) arguments in the definition string.

After assembling the macro definition string, the Macroassembler continues processing your source module, moving to the statement immediately following the macro call.

Assigning Locations

As the Macroassembler processes your source module, it assigns a memory location to each word of machine code it generates. The following discussion describes how the Macroassembler assigns memory addresses and how you may control this process.

Memory

Before explaining how MASM assigns addresses, we should review the characteristics of memory. AOS organizes memory into pages. Each page contains 1K 16-bit words (K equals $1,024_{10}$). The first page is called page zero; the second page is page one, etc.

The AOS operating system allows each process to occupy up to 32 pages or 32KW ($32,768_{10}$ words) of memory. The area of memory accessible to a process is that process's logical address space. For the Macroassembler's purposes, your logical address space is divided into two basic areas:

* Lower page zero (absolute and ZREL)

* NREL (shared and unshared)

Lower page zero contains locations 0 through 377_8 and is unshared (i.e., it is available to your process only). Locations 0-47_8 in lower page zero are absolute. These locations reserved for system- and hardware-specific data. Lower page zero relocatable (ZREL) memory extends from location 50_8 through 377_8.

NREL, or normal relocatable, memory extends from location 400_8 through 32KW-1, the upper boundary of your logical address space. NREL memory contains both shared and unshared memory pages. Shared pages are available to more than one process concurrently, while unshared pages are only accessible to your process. NREL memory also contains your process's system tables.

The following sections of this chapter explain the various parts of your logical address space and how to place object code in these areas of memory.

Note that the numeric memory boundaries we associate with ZREL and NREL are default values. You may alter these and other parameters at link time.

The previous discussion is a very brief overview of your logical address space. Refer to the AOS Programmer's Manual for further information about AOS memory organization.

Location Counter

When assigning memory locations to the words in your source, the Macroassembler manipulates an internal variable called the location counter. This variable holds the numeric address and relocation base (described below) of the next memory location that MASM will assign.

Use the value symbol period (.) to represent the location counter; thus, the expression

```
.+3
```

equals the current value of the location counter plus 3. Chapter 7 provides more information about this value symbol.

You may alter the value and relocation base of the location counter with pseudo-ops .GLOC, .LOC, .NREL, and .ZREL. We explain most of these pseudo-ops later in this chapter; detailed descriptions of each appear in Chapter 7.

Partitions

During pass one, the Macroassembler sorts the words in your source module into categories or partitions according to where in memory that code will reside. Each memory partition has certain attributes. The Macroassembler assigns a word of object code to the memory partition that matches its attributes. In general, the object code in a single memory partition will be contiguous in your executable program file. Figure 3-3 shows sections of source code sorted into different partitions.

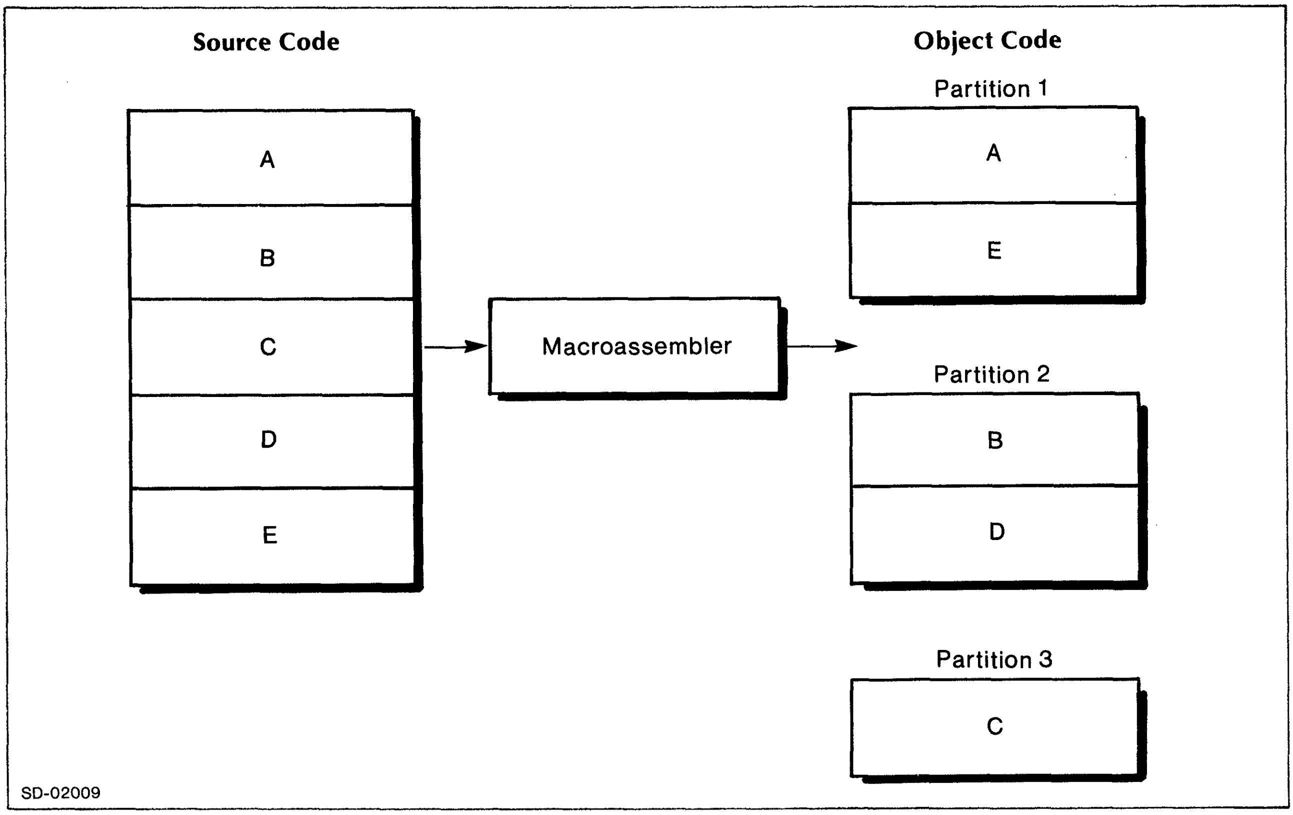


Figure 3-3. Sorting Code into Memory Partitions

Note that several sections of source code may contribute to the same partition. In Figure 3-3, code blocks A and E both reside in memory partition 1.

The assembly listing specifies the numeric address and memory partition of every word in your program. (See "Assembly Listing" in Chapter 4 for more information). The Macroassembler produces object code that will be place of the following partitions:

- absolute
- ZREL
- NREL unshared code
- NREL shared code

We discuss these partitions in the following sections. The _AOS Link User's Manual_, _AOS Library File Editor User's Manual_, and _AOS/VS Link and Library File Editor User's Manual_ also describe the various partitions.

A partition with the _absolute_ attribute contains object code that must reside at specific memory addresses. If you explicitly state the memory location of a word of code (e.g., with a .LOC pseudo-op whose argument is an absolute expression), the Macroassembler places that word in the absolute partition. Absolute words of code can reside at any location in memory (i.e., locations 0 through 32KW-1).

Partitions with the ZREL (lower page zero relocatable) and NREL (normal relocatable) attributes contain object code that is relocatable. Relocatable words of code need not reside at specific addresses. Instead, their addresses are relative to the addresses of other words. That is, relocatable words relate to and are dependent upon other words in your source code; they are not dependent on specific memory addresses.

The ZREL attribute defines words that must reside in the relocatable portion of lower page zero (locations 50_8 to 377_8). You may express any address in a ZREL partition in 8 bits. Thus, when referencing a location in a ZREL partition, you may use any memory reference instruction (MRI), since all have displacement fields of 8 or more bits.

Words in an NREL partition may reside anywhere in the address range 400_8 to 32KW-1.

You may address any word in a NREL partition with extended memory reference instructions. These instructions (i.e., ELDA and ESTA) have 15 bits in their displacement fields.

Object code can be shared or unshared. Code that is in the shared NREL partition can be executed by several processes at the same time. Code in the unshared NREL partition can be executed only by one process at a time. Code in the ZREL partition is always unshared.

Refer to the AOS Link User's Manual, the AOS/VS Link and Library File Editor User's Manual, and your operating system programmer's reference manual for more information about shared and unshared memory partitions.

The Macroassembler creates a partition only if you direct code to that partition. Therefore, each object module may not contain every partition type; it contains only the necessary ones.

The following sections describe each of the partitions in detail.

Absolute Partition

The absolute partition contains all object code that must reside at specific locations in memory. You can place code in the absolute partition by issuing the .LOC pseudo-op followed by an absolute expression. Generally, an absolute expression is one that does not contain any variables with relocatable values (we describe absolute expressions in detail later in this chapter). The absolute partition may reference any location in memory.

For example:

```
        .LOC            100
                 .
                 .
                 .
```

The Macroassembler begins placing the code that follows the .LOC statement at memory location 100. Similarly,

```
        A=50
        .LOC            A+100
                 .
                 .
                 .
```

directs the assembler to start placing object code at location 150 (the value of A plus 100). Note that the variable A has an absolute value (50).

The nature of absolute addressing is such that the words in the absolute partition are not contiguous in memory. For example, certain absolute code may begin at location 150 while other absolute code may reside at location 1025. This is the only partition that contains code that is not contiguous in memory.

In general, you do not store your source code in the absolute partition. Instead, you use the relocatable partitions (described in the following sections of this chapter).

If you must use absolute locations, be aware that relocatable code may overwrite your absolute code (or vice versa). For example, suppose you specify an absolute reference to location 100_8. If you include ZREL code in your program (relocatable code in locations 50_8 - 377_8), the Macroassembler may overwrite the absolute code at location 100.

The only way to prevent overwriting absolute locations is to redefine the partition boundaries such that the relocatable code cannot overlap the absolute area(s). For example, you may redefine ZREL to extend from locations 100_8 to 377_8 instead of from 50_8 to 377_8. Then, you may use locations 50_8 to 77_8 as absolute addresses and ZREL code will not overwrite them. Refer to the AOS Link User's Manual or the AOS/VS Link and Library File Editor User's Manual for information about how to change relocatable partition boundaries.

ZREL Partition

The ZREL partition contains object code that is relocatable but must reside in lower page zero. You may reference any location in ZREL memory in an 8-bit field. Thus, you may reference ZREL locations with any memory reference instruction. For this reason, you usually place pointers and frequently used data in the ZREL partition.

Use the .ZREL pseudo-op to indicate this partition. For example:

```
.ZREL
.            ;These words reside
.            ;in the
.            ;ZREL partition.
```

All code following the .ZREL pseudo-op resides somewhere in the relocatable portion of lower page zero (locations 50-377_8). In addition, this code is contiguous in memory.

NREL Partitions

The NREL partitions (shared and unshared) contain relocatable object code that may reside anywhere in the address range 400_8 to 32KW-1. To reference locations in the NREL partitions, you generally use memory reference instructions that have 15-bit displacement fields.

Use the .NREL pseudo-op to place code into either the shared or unshared NREL partitions. If you do not supply an argument to .NREL (or if the argument evaluates to 0) the words following the pseudo-op will reside in the unshared NREL code partition. For example:

```
.NREL
.           ;These memory words reside
.           ;in the unshared
.           ;NREL code partition.
```

To place words in the shared NREL code partition, pass a non-zero value to .NREL. For example:

```
.NREL   1
.              ;These words reside in
.              ;the shared
.              ;NREL code partition.
```

All the words in NREL partitions are contiguous in your program file. Thus, the assembler assigns sequential addresses to all words in the unshared NREL partition. Similarly, all code in the shared NREL partition receives sequential addresses. Note that the partitions themselves need not be contiguous in memory.

Relocatability

The previous sections describe memory partitions and their attributes. To review the main points, the Macroassembler groups words of code with similar attributes together into a partition. MASM usually creates several partitions, since all words in a module rarely have the same attributes.

All partitions, except the absolute partition, contain relocatable code. This code does not have to reside at specific addresses but may be anywhere within broad location ranges. Relationships between relocatable words are more important than specific locations.

Relocation Bases

As the Macroassembler places words from your module into the various partitions, it assigns each word a unique address within that partition. For words in the absolute partition, the assembler assigns the addresses you explicitly specify in your source module; all other partitions contain relocatable words whose addresses are not explicitly stated in the source.

The assembler assigns temporary locations (starting with zero) to the words in each relocatable partition. Each partition has its own address base, called a relocation base, and the locations in each partition start at temporary address 0. For example, the words in the ZREL partition receive contiguous addresses starting at 0; similarly, the words in the unshared NREL code partition also receive contiguous addresses starting with 0.

Table 3-1 illustrates how the assembler assigns addresses to the words in your source module. By default, each data entry occupies one word of memory. Thus, there is one address for each data statement in Table 3-1.

Table 3-1. Assigning Addresses Within Partitions

Source Code	Addresses	Partition
.TITL RELOC		
.ZREL		
20	0	ZREL
30	1	
.NREL 0		
40	0	Unshared NREL code
50	1	
60	2	
.NREL 1		
70	0	Shared NREL code
100	1	
.LOC 250		
110	250	
120	251	Absolute
.NREL 0		
130	3	Unshared NREL code
140	4	
.END		

During your assembly, you may leave a partition and later return to it (as in the case of the unshared NREL code partition in Table 3-1). When you return, the Macroassembler continues assigning addresses from the point where it left off.

From the above discussion, we can see that the assembler assigns consecutive addresses starting at 0 within each relocatable partition. These addresses serve as offsets from the partition's relocation base. The assembler cannot assign an absolute value to the relocation base because several separately assembled modules may specify the same memory partitions.

For example, two separately assembled modules may both place data in the ZREL memory partition. Since the assembler knows of only one module at a time, it always assigns ZREL locations starting with 0. Thus, each module will contain ZREL words with the same relative locations. Table 3-2 shows two separately assembled modules that both place code in the ZREL partition.

Table 3-2. Separately Assembled Modules With Similar Partitions

Source Module A	Relative Locations A	Source Module B	Relative Locations B
.TITL A .ZREL 20 30 40 .END	 0 1 2	.TITL B .ZREL 50 60 70 .END	 0 1 2

The Link utility can assign a value to each partition's relocation base because it has access to all the partitions and locations you use in the program. Since all partitions have the normal base attribute, Link assigns values to the relocation bases in such a way that similar partitions from different modules are contiguous in memory. Figure 3-4 illustrates how Link assigns addresses to similar predefined partitions from the separately assembled modules A, B, and C.

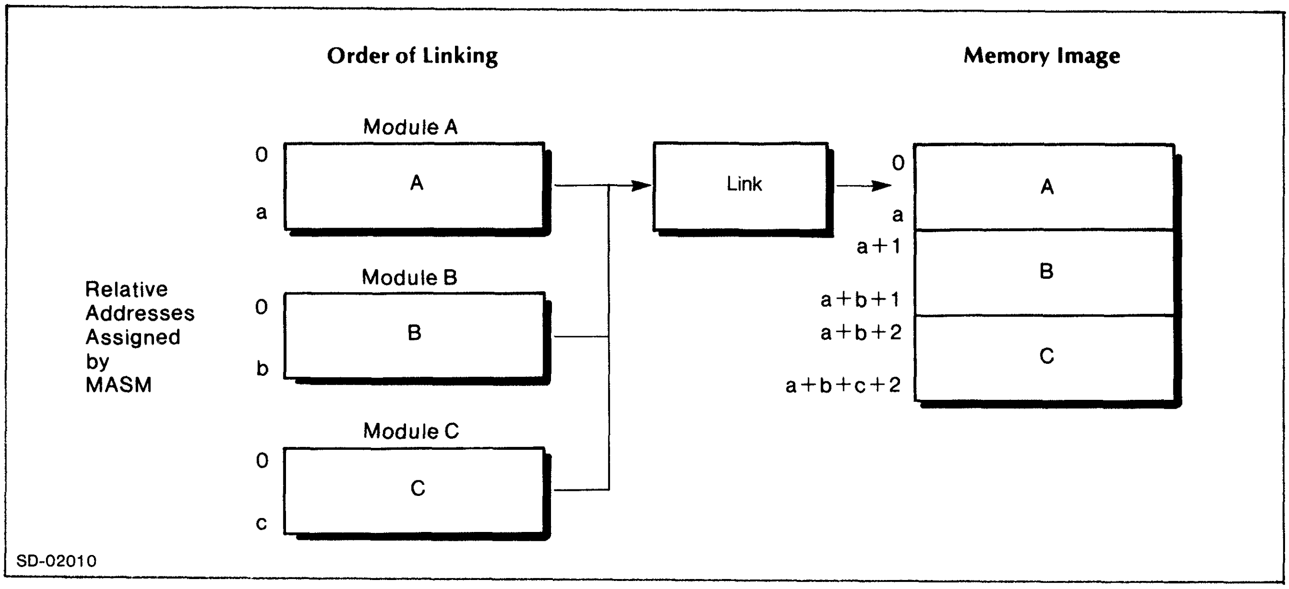


Figure 3-4. Linking Modules with Similar Partitions

The Link utility calculates each word's new location by adding the offset assigned by the Macroassembler to the relocation base assigned by Link. Thus, the Link utility maintains the relationships between words within a partition; i.e., contiguous words in a partition are also contiguous in the program file that Link produces.

The AOS Link User's Manual and AOS/VS Link and Library File Editor User's Manual provide more information about how Link assigns values to the relocation bases.

Relocation Bases and Symbols

Thus far, we have discussed relocation bases only with respect to partitions and addresses. However, the value of each symbol you use in your program also has a relocation base associated with it.

To review, there are four major relocation bases associated with the four memory partitions:

- Absolute
- ZREL
- Unshared NREL code
- Shared NREL code

The Macroassembler assigns each partition a unique relocation base. In addition, each symbol that you declare as external (i.e., with a .EXTD or .EXTN statement) also receives a unique relocation base.

The Macroassembler defines each symbol's value relative to a relocation base. For example,

```
        .ZREL           ;The following words reside in
                        ;the ZREL partition.
A:      10              ;Each entry requires one word of
B:      20              ;storage.  Thus, the value 10 resides
                        ;at relative location 0 and the value 20
                        ;resides at relative location 1.
```

The Macroassembler evaluates the label B relative to the ZREL relocation base:

$$B = RB_Z + 1$$

where:

RB_Z is the ZREL relocation base

1 is the offset from RB_Z that the Macroassembler assigns to B

When you use a symbol, you associate it with a relocation base either explicitly or implicitly.

If a symbol appears in an .EXTD or .EXTN statement, the assembler assigns it a unique relocation base. The symbol receives its relocation base explicitly because you directly associate the symbol with the base when you issue the pseudo-op.

Other symbols receive relocation bases implicitly through their association with a partition or another symbol. For example, a label always receives the relocation base of the partition in which it appears.

The following list summarizes the ways that the assembler can implicitly assign a relocation base to a symbol:

* If you associate a symbol with an address in a partition (e.g., a label), then that symbol's value receives that partition's relocation base. For example:

```
        .NREL     ;Unshared NREL partition.
X:      5         ;X has the same relocation base as
                  ;the unshared NREL partition.
        A=.+2     ;A is defined with respect to the
                  ;location counter and, thus, has the
                  ;same relocation base as the unshared
                  ;NREL partition.
```

* If you define one symbol with respect to another symbol that has a relocation base, the first symbol gets the second symbol's relocation base. For example:

```
        .EXTD M    ;M receives a unique relocation base.
        .NREL      ;Unshared NREL partition.
X:      5          ;X has the unshared NREL base.
        A=X+3      ;A has the same relocation base as X.
        N=M+1      ;N has the same relocation base as M.
```

* If you define a symbol in terms of integers alone, the symbol receives the absolute relocation base. For example:

```
        B=3             ;B has the same relocation base
                        ;as 3 (i.e., absolute base).
```

* All symbols defined by instructions have the absolute relocation base. For example:

```
    F=LDA 0,0    ;F has the absolute relocation base.
```

* All value symbols, except .LOC and period (.), have the absolute relocation base.

The assembly listing indicates both the value and the relocation base for each number, symbol, and expression you use in your program (see "Assembly Listing" in Chapter 4).

Absolute Addresses Versus Absolute Values

When discussing symbol relocation, we distinguish between absolute values and absolute addresses. A symbol with an absolute value keeps that value through the assembly, Link, and runtime processes. An absolute value must be an integer. It is exactly equal to the value you assign it in your source module. For example:

```
        A=10                        ;Symbols A, B, C, and D
        B=A+4                       ;all have absolute values.
        C=LDA       0,0
        D=.RDX
```

MASM can completely resolve all symbols with absolute values.

Absolute addresses represent specific address values in your logical address space. For example:

```
        .LOC            100
A:      0
```

Symbol A's value represents the address of the 100th word in your logical address space. However, MASM does not simply store A's value as 100. Rather, A's internal representation contains information required for address resolution at link time and runtime.

Since MASM cannot determine where in AOS memory your logical address space will reside, it cannot determine what A's runtime value will be. Thus, MASM cannot assign to any address, absolute or otherwise, a constant, integer value.

Suffice it to say that all locations have relocatable values at link time and runtime, even addresses in the absolute partition. Thus, all labels have relocatable values. This point is important when we discuss the relocation properties of expressions in the next section.

Relocation Bases and Expressions

Chapter 2 introduced expressions and explained the various operators. The following sections examine a different aspect of expressions: their relocation properties.

As we have seen, each symbol has a relocation base associated with its value. Similarly, MASM assigns a relocation base to each expression in your source.

There are two types of expressions:

- Absolute expressions
- Relocatable expressions

The following sections describe the properties of these two expression types.

Absolute Expressions

An absolute expression has an absolute value; that is, MASM can completely resolve the expression to an integer value.

The simplest absolute expressions contain operands that have integer values. For example, the following are all absolute expressions. (Remember that all value symbols except .LOC and period (.) have the absolute relocation base.)

```
5    6*3     (.PASS)       377!177    (.RDX)<=(6/2)
```

Similarly, the following are also absolute expressions if A and B have absolute values (e.g., A=10, B=.RDX):

```
A       A+150           B*.PASS          A+B*(A-10)
```

All the absolute expressions we've presented thus far contain operands that have absolute values. However, absolute expressions can also contain relocatable operands if the resulting value has no relocatable components. That is, if all relocatable components cancel each other out, the expression is absolute. Consider the following example:

```
        .ZREL
A:      10
B:      20
        (B-A)+40
```

Earlier in this chapter ("Relocation Bases and Symbols"), we showed how MASM evaluates each symbol's value with respect to a relocation base. Thus, MASM computes the values for A and B as follows:

$$A = RB_Z + 0$$

$$B = RB_Z + 1$$

where:

RB_Z is the ZREL relocation base

0 is the offset from RB_Z that MASM assigns to A

1 is the offset from RB_Z that MASM assigns to B

The values for A and B are relative to the ZREL relocation base. Thus, A and B have relocatable values.

MASM evaluates the expression (B-A)+40 as follows:

$$\begin{aligned}(B-A)+40 &= ((RB_Z + 1) - (RB_Z + 0)) + 40 \\ &= (RB_Z - RB_Z) + (1 - 0) + 40 \\ &= (0) + (1) + 40 \\ &= 41\end{aligned}$$

In this expression, the two relocatable components (RB_Z) cancel each other out, leaving an absolute value (i.e., 41). Thus, the expression (B-A)+40 is an absolute expression, even though it contains relocatable operands.

As we mentioned earlier, all labels have relocatable values. Thus, you may never use labels in absolute expressions unless their relocatable components cancel out.

As we have seen, MASM can completely resolve absolute expressions. That is, absolute expressions resolve to integer values at assembly time. Thus, you generally use absolute expressions when a value is required during the assembly process (e.g., index and accumulator arguments to memory reference instructions, and certain pseudo-op arguments).

In addition, since MASM can completely resolve an absolute expression, it verifies that the expression's value is legal for the field it appears in. For example:

```
        LDA             0,23571,0
```

The absolute expression 23571 is too large to fit in the 8-bit LDA displacement field. Thus, MASM returns an error for this instruction.

Relocatable Expressions

Relocatable expressions resolve to relocatable values. That is, the result of a relocatable expression is not simply an integer; it contains a relocatable component that cannot be resolved until link time.

All relocatable expressions conform to one of the following formats:

$$\underline{\text{<2*>}}\textbf{rel-symbol} \qquad \textbf{abs-expr} \left\{ \begin{matrix} + \\ - \end{matrix} \right\}$$

or

$$\textbf{abs-expr} \left\{ \begin{matrix} + \\ - \end{matrix} \right\} \textbf{rel-symbol}\underline{\text{<*2>}}$$

where:

rel-symbol is a symbol whose value is relocatable (see "Relocation Bases and Symbols" earlier in this chapter)

abs-expr is an absolute expression

According to this definition, the following module contains several relocatable expressions:

```
   .ZREL
A: 10              ;A has the ZREL relocation base.
   .NREL
B: 20              ;B and C have the unshared NREL
C: 30              ;relocation base.
   A+20            ;Relocatable symbol A plus absolute expression 20.
   B-5             ;Rel-symbol B minus abs-expr 5.
   A+(B-C)         ;Rel-symbol A plus abs-expr (B-C).
   (10+(B-C))-A    ;Abs-expr (10+(B-C)) minus rel-symbol A.
```

Note that you may include more than one relocatable symbol in an expression as long as all but one of their relocation bases cancel out. In the example module, A+(B-C) contains three relocatable values. However, B and C have the same relocation base (unshared NREL); thus, (B-C) has an absolute value (see "Absolute Expressions" for more information).

Also, remember that MASM assigns each external symbol a unique relocation base. This is true even if you declare several symbols in the same statement; for example:

```
        .EXTD           X,Y
```

X and Y receive different relocation bases. Since their bases are unique, you cannot cancel either base out, as you could with symbol B and C in the previous example. Thus, you may never use two external symbols in the same expression.

The relocatable expression syntax shows that you may multiply the relocatable symbol by two. In the following example, both 2*X and (10)+(2*Y) are legal expressions.

```
        .ZREL
X:10
Y:20
        2*X
        (10)+(2*Y)
```

An expression whose relocatable symbol value is multiplied by two is called byte-relocatable. In most cases, you use byte-relocatable expressions as byte pointers (values that specify a byte's address). In the example module, the expression 2*X is a byte pointer to the byte starting at address X. For more information on byte pointers, refer to the Programmer's Reference, ECLIPSE®-Line Computers manual.

MASM cannot completely resolve relocatable expressions, since relocation bases do not receive values until link time. Thus, MASM cannot determine whether a relocatable expression's value is legal for the corresponding field. For example:

```
        .NREL                ;Unshared NREL.
X:      10                   ;Symbol X is relocatable.
        .NREL     1          ;Shared NREL.
        LDA       0,X+50     ;Load the value starting at the 50th
                             ;word after address X into AC0.
```

The LDA instruction provides a 8-bit field for the displacement value. However, since X does not have an absolute value at assembly time, MASM cannot determine whether the expression X+50 can fit into a 8-bit field.

In summary, when using a relocatable expression, be sure that its value can be represented in the corresponding field. If it cannot, you will receive an error when Link resolves the expression.

Resolving Relocatable Expressions

The previous discussion explained how to create and use relocatable expressions in your source module. This section describes how MASM evaluates relocatable expressions.

At assembly time, all relocatable expressions must resolve to a relocatable component, an unresolved operator, and an integer component (i.e., an absolute value). In addition, the unresolved operator must be either + or -.

An example will help clarify these rules:

```
        .ZREL    ;ZREL memory partition.
Y:      5        ;The value 5 resides at location 0
X:      15       ;in this partition; 15 resides at
        X+4      ;location 1.
```

In this example, the value of label X equals the ZREL relocation base plus 1 word. Therefore, X's value equals the second address in the ZREL partition. You can think of X's value as

$$X = RB_Z + 1$$

where:

RB_Z is the ZREL relocation base

1 is the offset from RB_Z that MASM assigns to X

The Macroassembler cannot completely resolve the expression X+4 because the ZREL relocation base does not have a value. However, the Macroassembler can partially evaluate the expression as follows:

$$(X+4) = ((RB_Z + 1) +4)$$

$$= (RB_Z + 5)$$

At this point, the Macroassembler cannot process the expression any further. Thus, it passes Link the absolute value 5 (integer component), the ZREL relocation base RB_Z (relocatable component), and the unresolved operator +.

During the Link process, the ZREL relocation base receives a value. Then, Link can fully resolve the values for the symbol X and the expression X+4.

In most cases, you include only one relocatable operand in each expression (as in the example). The Macroassembler does, however, allow you to include more than one relocatable value in a single expression. Again, the expression must resolve to a single relocation base, an operator, and an integer component or you will receive an error. For example:

```
        .ZREL
Y:      10
X:      20
        .NREL
W:      30
Z:      40
        (X-Y)+Z
```

The expression (X-Y)+Z includes three relocatable operands. X and Y have the ZREL relocation base; Z has the unshared NREL base. The Macroassembler evaluates this expression as follows:

$$
\begin{aligned}
(X-Y)+Z &= ((RB_z+1)-(RB_z+0))+(RB_n+1)\\
&= (RB_z-RB_z)+(1-0))+(RB_n+1)\\
&= ((0)+(1))+(RB_n+1)\\
&= RB_n+2
\end{aligned}
$$

RB_z is the ZREL relocation base, and RBn is the unshared NREL base. The values 1, 0, and 1 are the offsets for X, Y, and Z from their respective relocation bases.

Since both ZREL relocation bases cancel out, the expression is legal. After processing the expression, MASM passes Link the absolute value 2 (integer component), the relocatable value RB_n (relocatable component), and the unresolved operator +.

The previous section explained that you may multiply a relocatable symbol value by 2 to create a byte-relocatable expression; for example:

```
        .ZREL
Y:      10
X:      10
        2*X
```

The expression 2*X serves as a byte-pointer to the first byte at address X. MASM evaluates this expression as follows:

$$
\begin{aligned}
2*X &= 2 * (RB_z + 1) \\
&= (2 * RB_z) + (2 * 1) \\
&= (2 * RB_z) + 2
\end{aligned}
$$

MASM cannot process this expression any further. Thus, it passes Link the relocatable component $2*R_z$, the unresolved operator +, and the integer value 2. Any expression whose relocatable component equals two times a relocation base is byte-relocatable.

Table 3-3 displays different forms of expressions and shows how the assembler resolves each. We use the following notation in Table 3-3:

n and m — represent two different absolute values; they may be integers, symbols, or absolute expressions

r and p — represent two relocatable values with the same relocation base; they may be symbols or expressions

s — represents a relocatable value whose relocation base is different from that of r and p

RB_r — is the relocation base associated with r's value

$r_{(off)}$ — is the offset of r's value from relocation base RB_r; i.e., $r = RB_r +$ r (sub<(off)>)

All expressions involving the operators <, <=, >, >=, ==, or <> result in an absolute value of either zero (false) or one (true). When operands in these expressions have different relocation bases, all comparisons result in a value of zero (false), except when the operator is <> (not equal to).

You cannot use relocatable operands with the logical operators & and !.

Table 3-3. Relocatable Expressions

Expression	Relocatable Component	Integer Component	Unresolved Operator(s)
n+m	absolute	n+m	none
n-m	absolute	n-m	none
n*m	absolute	n*m	none
n/m	absolute	n/m	none
n<=m	absolute	n<=m	none
n&m	absolute	n&m	none
n!m	absolute	n!m	none
n+r	absolute	$n+r_{(off)}$	+
n-r	RB_r	$n-r_{(off)}$	-
r-n	RB_r	$r_{(off)}-n$	+
r+n	RB_r	$r_{(off)}+n$	+
2*r	$2*RB_r$	$2*r_{(off)}$	+
r-r	absolute	0	none
r-p	absolute	$r_{(off)}-p_{(off)}$	none
n/r	**********************ILLEGAL********************		
n*r	**********************ILLEGAL********************		
r+r	**********************ILLEGAL********************		
r*r	**********************ILLEGAL********************		
r&r	**********************ILLEGAL********************		
n&r	**********************ILLEGAL********************		
r!r	**********************ILLEGAL********************		
n!r	**********************ILLEGAL********************		
r+p	**********************ILLEGAL********************		
r*p	**********************ILLEGAL********************		
s+r	**********************ILLEGAL********************		
s-r	**********************ILLEGAL********************		
s*r	**********************ILLEGAL********************		
r/p	**********************ILLEGAL********************		

NOTE: Any expression with a relocatable component equal to $2*RB_r$ is byte-relocatable.

Resolving Locations in Memory Reference Instructions

Chapter 2 ("Assembly Language Instructions") briefly mentioned memory reference instructions (MRIs) and their major uses. Memory reference instructions let you access locations in your logical address space.

Each memory reference instruction requires you to specify a unique location in memory. You may do this in one of two ways:

* you can explicitly supply a displacement value and an addressing index; or

* you can supply a single address value and let MASM calculate the appropriate displacement and index values

These two addressing methods provide you with considerable programming power and flexibility. The following sections of this chapter explain how to use these two methods.

We do not describe indirect addressing in this section. Refer to "At Sign (@)" in Chapter 2 for a description of how MASM assembles memory reference instructions containing the indirect addressing indicator @. The Programmer's Reference, ECLIPSE®-Line Computers manual has more information on indirect addressing. It also describes each memory reference instruction in detail.

Supplying Both a Displacement and an Index

The ECLIPSE computers' memory reference instructions (MRI) provide four ways to address locations in memory:

* absolute addressing

* program counter (PC) relative addressing

* accumulator (AC) relative addressing using AC2

* accumulator (AC) relative addressing using AC3

In absolute addressing, the Macroassembler takes the value you specify in the MRI's displacement field as an address in your logical address space.

In PC relative addressing, the Macroassembler uses the displacement value in the MRI as an offset from the instruction's address. That is, the Macroassembler computes the memory address by adding the displacement value to the address of the instruction.

In AC relative addressing, the Macroassembler computes the memory address by adding the displacement value to the contents of an accumulator (either AC2 or AC3). In other words, the displacement value serves as an offset from the value in an accumulator.

You may indicate one of these addressing modes by placing a value from 0 through 3 in an memory reference instruction's optional index argument (sometimes called the mode argument). Table 3-4 shows the four index values.

Table 3-4. MRI Index Values

Index Value	Addressing Mode
0	Absolute addressing
1	PC relative addressing
2	AC2 relative addressing
3	AC3 relative addressing

The next two examples illustrate the use of the index argument in memory reference instructions.

Our first example specifies PC relative addressing. When MASM calculates a PC relative address, it uses the value in the displacement field as an offset from the field's address. Since the displacement field often begins at the second word of the instruction, choose your displacement value accordingly.

```
LDA    0,3,1
50
100
200
```

The LDA instruction loads accumulator 0 (AC0) with the value that begins three words after the location of the displacement field (i.e., displacement value of 3; index value of 1). The LDA instruction is one word. The value 200 begins three words after the instruction. Therefore, after the LDA instruction, AC0 contains the value 200.

The second example shows the use of the absolute addressing index:

```
 .LOC        110
 4                        ;The value 4 resides at
     .                    ;absolute location 110.
     .
     .
LDA        1,110,0        ;Load the value at absolute
                          ;location 110 into AC1.
```

The LDA index value of 0 directs MASM to use the displacement value as an absolute address (not as an offset from the PC or an AC). Thus, the Macroassembler loads the value at address 110 into AC1.

Thus far, all our examples have used absolute displacement values. If you specify a relocatable value in a memory reference instruction that has an 8-bit displacement field, you may not supply an addressing index of 2 or 3. That is, you may not specify AC relative addressing with 8-bit relocatable displacement values. The following example illustrates this rule:

```
     .ZREL
R:   3                  ;R's value is relocatable (ZREL).
     A=3                ;A's value is absolute.
          .
          .
          .
     LDA      0,A,1    ;Legal.
     LDA      0,R,0    ;Legal.
     LDA      0,R,2    ;Illegal.
     ELDA     0,R,2    ;Legal (ELDA has 15-bit displacement field).
```

The first LDA instruction contains an absolute displacement value, A. The second and third instructions specify relocatable displacement values (i.e., R). The third statement is in error because it also specifies an index value of 2 (AC2 relative addressing). The last statement is legal, because the ELDA instruction has a 15-bit displacement field. Again, you may not specify index values of 2 or 3 when using 8-bit relocatable displacement values.

When specifying a displacement, be sure that value can fit into the corresponding field of the MRI instruction. The 16-bit ECLIPSE MRI instructions have displacement fields of either 8 or 15 bits). Table 3-5 shows the legal range of the displacement value for different index modes.

Table 3-5. MRI Displacement Values

Index Value	Length of Displacement Field	
	8 bits	**15 bits***
0 (absolute addressing)	0 to 377_8 or 0 to 255_{10}	0 to 77777_8 or 0 to $32,767_{10}$
1, 2, 3 (PC or AC relative addressing)	-200 to $+177_8$ or -128 to 127_{10}	-40000 to $+37777_8$ or $-16,384$ to $+16,383_{10}$

*Note that you may reference any location in your logical address space with a 15-bit displacement field.

Supplying Only a Displacement Value

The previous discussion showed how you may identify a location by specifying a displacement value and an addressing index. Alternatively, you may supply a single address argument and let MASM compute the appropriate displacement value and addressing index.

Memory reference (8-bit displacement) and extended memory reference (15-bit displacement) instructions can both be coded without an index field. The Macroassembler computes an effective location differently, depending on which kind of instruction is used.

Resolving Locations for Standard Memory Reference Instructions

The index mode field in memory reference instructions is optional. When the Macroassembler encounters a memory reference instruction with no index mode specified, it examines the instruction's address expression. If the address is in page zero (0 through 377_8), the Macroassembler sets the instruction's index mode bits to 00. The instruction's displacement field is set as follows:

1. If the address is absolute, the displacement field is set to the value of the address.

2. If the address is page zero relocatable (assembled with the .ZREL pseudo-op), the displacement field is set to the value of the address (with page zero relocation). The source line's data field location flag (column 16 in the listing) is set to - (the page zero relocation flag).

3. If the address is an external displacement (assembled with the .EXTD pseudo-op), the instruction's displacement field is set to zero. The line's data field location flag is set to $ (the external definition flag).

If the instruction's address is within 177_8 words of the location counter, the instruction's index mode bits are set to 01. Therefore, address resolution is based on the current contents of the location counter. The instruction's displacement field is set to the value of the address minus the contents of the location counter.

If the instruction's address, or the resolution of displacement to an address, does not produce an effective address within the proper range, an addressing (A) error is reported.

Resolving Locations for Extended Memory Reference Instructions

Extended memory reference instructions assemble into two 16-bit words. The first word specifies the instruction and index. The second word specifies the displacement and whether indirect addressing is being used. Extended memory reference instructions can also be coded without an explicit index mode. The Macroassembler proceeds as described below when it encounters this kind of instruction.

1. The instruction's index mode bit is set to 01 if the location counter and addressed location have the same address type. In other words, both addresses must be ZREL, shared NREL, or unshared NREL.

2. The instruction's index mode bit is set to 00 if the location counter and addressed location do not have the same address types.

3. The instruction's index mode bit is set to 01 if the addressed location is externally defined. In this case, the Macroassembler must make an assumption about the actual location type of the destination symbol. The Macroassembler assumes that the resolved address of an EDSZ, EISZ, EJMP, EJSR, ELDA, ESTA, or ELEF will be determined by word relocation rather than byte relocation. Only if the object of an ELEF is byte-relocatable could the Macroassembler's assumption be incorrect. In this case, you should force absolute addressing by coding an index mode of 0 into the ELEF instruction.

As we mentioned earlier, you must ensure that the displacement value will fit into the field provided by the MRI instruction. Refer to Table 3-5 for the legal displacement value ranges for absolute and PC relative addressing.

Do not code a skip instruction immediately before an extended instruction. If the skip occurs, the program branches to the extended instruction's second word. The Macroassembler will generate a questionable line (Q) error if you violate this rule.

Using Literals in Memory Reference Instructions

All memory reference instructions must specify an address field. This address is used to:

1. Access the contents of the memory location specified in a LDA or ELDA instruction.

2. Modify the memory location specified in the STA, ISZ, or DSZ family of instructions.

3. Transfer control when a JMP or JSR instruction is used.

Often, you only want to specify the contents of a memory location without concern for its address. Such a specification is called a literal reference, or simply a literal.

You can use literals with all memory reference instructions. The Macroassembler dumps your program's literals and assigns memory locations starting at the first .ZREL location that is available after pass 1. Therefore, all literal references are directly addressable. You can use the .NLIT pseudo-op to assign literals to the NREL partition. The .NLIT pseudo-op must be used with the .LPOOL pseudo-op. .LPOOL dumps the currently defined literals into a data block. These pseudo-ops are described in Chapter 7.

The syntax of a literal reference is as follows:

$$\textbf{memory-reference}\square\underline{\langle AC,\rangle}\textbf{=} \quad \left\{ \begin{array}{c} \textbf{expression} \\ \textbf{instruction} \end{array} \right\}$$

Note that a literal may be any expression or instruction.

Literals are frequently used to load a constant into an accumulator. For example:

```
      LDA 1,=3
```

loads AC1 with the value 3.

Expressions are also acceptable. For example:

```
      LDA 0,=1B0+"A/2
```

loads AC0 with the value 40040.

Instructions are also acceptable. For example:

```
    LDA 1,=SUBZ# 2,3,SNC
```

loads AC1 with the assembled value of the SUBZ instruction (156433).

The previous examples use absolute expressions as literals. However, any relocatable expression is legal. For example:

```
               .NREL
                .
          A:    .
                .
               LDA 2,=A
```

loads the value of A into AC2. You can also use a literal to form a byte pointer to a text string labeled TX as shown below.

```
               LDA 1,=2*TX
                .
                .
         TX:   .TXT "TEXT STRING"<nl>
```

Literal labels let you communicate with subroutines without concern for addressing errors. The following statement calls subroutine SUBR, regardless of whether SUBR is directly addressable.

```
      JSR   @=SUBR
```

End of Chapter

Chapter 4
Output from the Macroassembler

The Macroassembler can produce five different types of output during the assembly process:

* Object file
* Assembly listing
* Cross-reference listing
* Error listing
* Permanent symbol table

All these forms of output, except the error listing, are optional; the Macroassembler always reports assembly errors.

The <u>permanent symbol table</u> defines symbols for use in future assemblies. It usually resides in disk file MASM.PS (MASM16.PS in AOS/VS MASM16) and contains definitions for all operating system calls and system parameters.

If you produce a permanent symbol table, you may also generate an assembly listing, but the listing will contain only assembly errors. Refer to Chapter 8 for information about how the Macroassembler uses the permanent symbol table and how you can build one.

The following sections of this chapter describe the other four types of Macroassembler output.

Object File

The <u>object file</u> is a binary translation of the code in your source file. Each line of source code translates into a binary number that is a multiple of 16 bits (one word) in length. The Macroassembler assigns each one-word number an address (which may be absolute or relocatable). Chapter 3 describes how the Macroassembler assigns addresses and performs other operations necessary to produce the object file.

The Macroassembler does not normally produce an object file if your source contains errors. If you want the Macroassembler to produce an object file, even if there are assembly errors, include the /R function switch on the MASM command line.

You can use the /N switch to tell the Macroassembler not to create an object (.OB) file. You usually use the /N switch to locate errors in your source code.

Normally, the object file receives the same name as the first source module on the MASM command line without the .SR extension, if any, and with the new extension .OB. If you include either the /B= switch on the MASM command line or the .OB pseudo-op in your source module, the Macroassembler overrides the default-naming convention. Table 4-1 shows the hierarchy that the Macroassembler uses to name object files.

Table 4-1. Object Filename

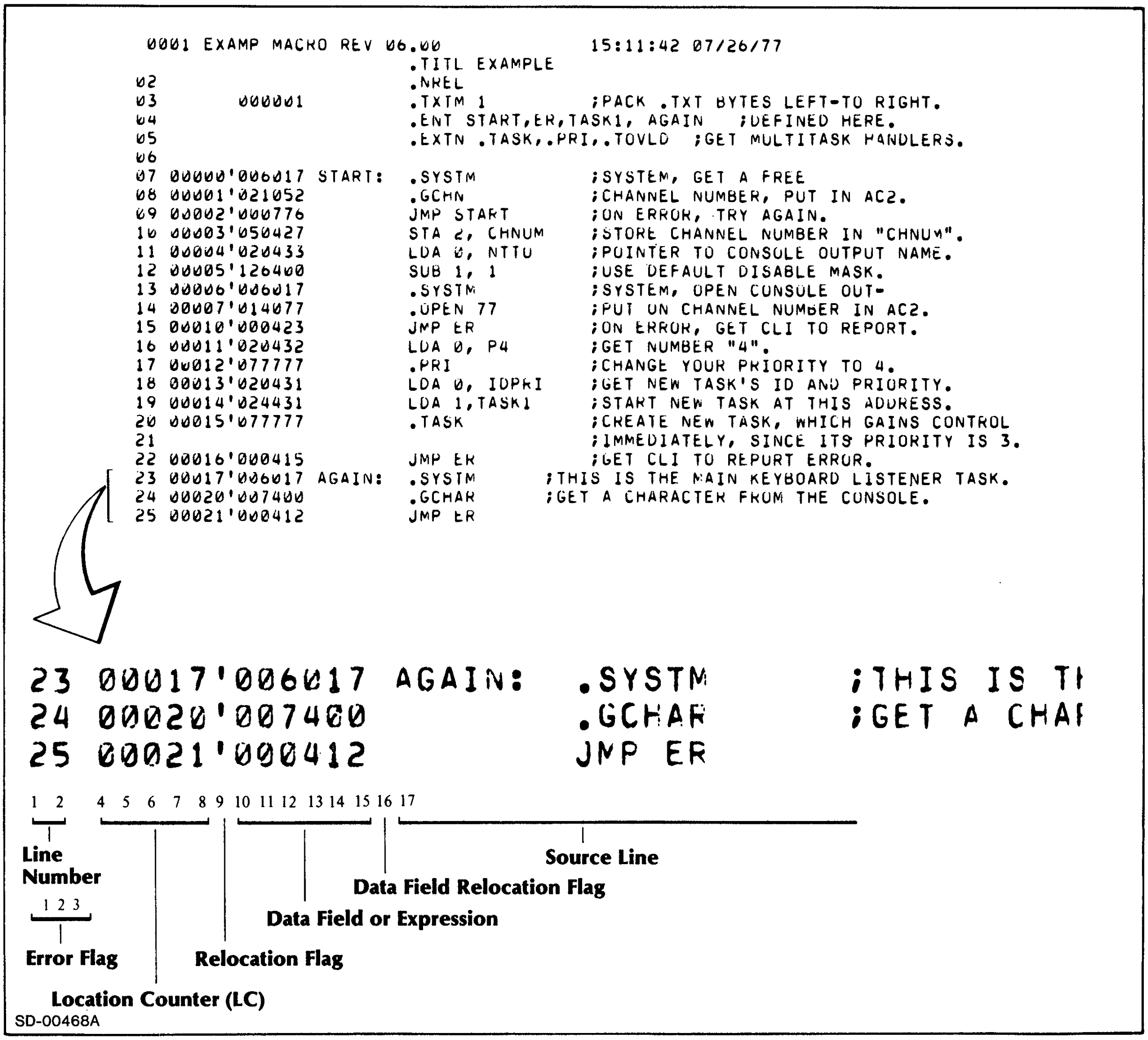
Priority	Object Filename	Description
1 (highest)	/B=filename	Function switch on the MASM command line
2	.OB}filename	Pseudo-op in a source module
3 (lowest)	Default name	Name of the first source module on the MASM command line

The object file is not executable; you must process it with the Link utility to produce a program file. Chapter 8 provides the general Link command line; the AOS Link User's Manual and the AOS/VS Link and Library File Editor User's Manual describe the Link utilities in detail.

Assembly Listing

The information in the assembly listing shows you how the Macroassembler interpreted your source file. The listing consists of a series of lines. Each line is divided into various fields.

Figure 4-1 shows the fields in a sample assembly listing. Table 4-2 lists the fields and their contents.



```
 0001 EXAMP MACRO REV 06.00                 15:11:42 07/26/77
                            .TITL EXAMPLE
02                          .NREL
03       000001             .TXTM 1          ;PACK .TXT BYTES LEFT-TO RIGHT.
04                          .ENT START,ER,TASK1, AGAIN   ;DEFINED HERE.
05                          .EXTN .TASK,.PRI,.TOVLD  ;GET MULTITASK HANDLERS.
06
07 00000'006017 START:      .SYSTM           ;SYSTEM, GET A FREE
08 00001'021052             .GCHN            ;CHANNEL NUMBER, PUT IN AC2.
09 00002'000776             JMP START        ;ON ERROR, TRY AGAIN.
10 00003'050427             STA 2, CHNUM     ;STORE CHANNEL NUMBER IN "CHNUM".
11 00004'020433             LDA 0, NTTO      ;POINTER TO CONSOLE OUTPUT NAME.
12 00005'126400             SUB 1, 1         ;USE DEFAULT DISABLE MASK.
13 00006'006017             .SYSTM           ;SYSTEM, OPEN CONSOLE OUT-
14 00007'014077             .OPEN 77         ;PUT ON CHANNEL NUMBER IN AC2.
15 00010'000423             JMP ER           ;ON ERROR, GET CLI TO REPORT.
16 00011'020432             LDA 0, P4        ;GET NUMBER "4".
17 00012'077777             .PRI             ;CHANGE YOUR PRIORITY TO 4.
18 00013'020431             LDA 0, IDPRI     ;GET NEW TASK'S ID AND PRIORITY.
19 00014'024431             LDA 1,TASK1      ;START NEW TASK AT THIS ADDRESS.
20 00015'077777             .TASK            ;CREATE NEW TASK, WHICH GAINS CONTROL
21                                           ;IMMEDIATELY, SINCE ITS PRIORITY IS 3.
22 00016'000415             JMP ER           ;GET CLI TO REPORT ERROR.
23 00017'006017 AGAIN:      .SYSTM       ;THIS IS THE MAIN KEYBOARD LISTENER TASK.
24 00020'007400             .GCHAR       ;GET A CHARACTER FROM THE CONSOLE.
25 00021'000412             JMP ER
```

Figure 4-1. Sample Assembly Listing

Table 4-2. AOS MASM Assembly Listing Fields

Columns	Information Contained
1-3	If the assembler finds no errors in your source, columns 1-3 contain a two-digit line number followed by a space. If there are input errors, each error generates a single letter code. Only three error codes can be listed on a line. The first error generates a letter code in the third column. The second error code is in column two, and the third error code is in column one. Source lines that have errors receive no line number. We describe the AOS MASM error codes in Appendix C.
4-8	The Macroassembler displays the value of the location counter in these five columns. The location counter contains the address of the first word of the assembled source statement. The columns will be blank if the source statement does not generate any storage words.
9	This column contains a flag character specifying the current addresses' relocation mode (see Table 4-3).
10-15	This is the data field of the assembled instruction or expression; or the value of an assignment statement or pseudo-op argument. In all other cases, these columns are blank.
16	This field contains a flag character specifying the data field's relocation base (see Table 4-4, shown later).
17...	The source statement exactly as written (except for macro expansions).

Column 9 contains a one-character symbol indicating the relocation base (partition) of the address. Table 4-3 lists the flags and their meanings.

Table 4-3. Location Counter Relocation Symbols

Symbol	Relocation Base
(spaces)	Absolute
-	Page zero relocatable (ZREL)
'	Unshared NREL
!	Shared NREL

Refer to Chapter 3 for a description of how the Macroassembler assigns locations and relocation bases to the statements in your source file.

Following the address relocation symbol is a 6-column field. This field contains the assembled value of the first 16-bit word in the current source statement. A two-word instruction uses two of these fields. The following example shows how the MASM listing represents one- and two-word instructions:

```
01 00034'143000        ADD     2,0     ;One-word instruction
02 00052'163770        ADDI    -7,0    ;Two-word instruction
03       177771
```

Certain source statements do not generate storage words in your object file. For these source lines, the listing data field contains the value of an argument or other relevant expression. For example:

```
01       000001       .NREL   1
```

The data field (columns 10-15) contains the value of the argument to .NREL (i.e., 1).

A one-character symbol indicating the relocation base of the data value follows the data field. The data field relocation symbols are shown in Table 4-4. Chapter 3 describes how the Macroassembler assigns a relocation base to a value in your source.

Table 4-4. Data Field Relocation Symbols

Symbol	Relocation Base
space	Absolute
-	Page zero relocatable
=	Page zero, byte-relocatable
'	Unshared code
!	Shared code
"	Unshared code, byte-relocatable
&	Shared code, byte-relocatable
$	Displacement field is externally defined

The last item on each line of the assembly listing is the original ASCII source line. The listing gives your source line exactly as you entered it, except that it includes macro expansions in the appropriate places.

When the Macroassembler outputs an assembly listing, you usually receive a cross-reference listing of symbols in the same file. See "Cross-Reference Listing" in this chapter for more information.

In addition, the Macroassembler reports assembly errors at the beginning of any source line that contains an error. Refer to "Error Listing" later in this chapter for more information about how the Macroassembler reports assembly errors. Appendix C lists the AOS MASM error codes.

Assembly Listing Control

The Macroassembler does not produce an assembly listing automatically. If you want one, you must include either the /L or /L=filename switch on the MASM command line. The /L switch directs the Macroassembler to send the assembly listing to the generic file @LIST; the /L=filename switch sends the listing to the specified file.

The Macroassembler provides two tools that allow you to manipulate the contents and format of the assembly listing:

Listing control pseudo-ops:	.EJEC, .NOCON, .NOLOC, .NOMAC, and .RDXO
Listing suppression indicator:	two asterisks (**)

These features do not alter the object file; they affect only the assembly listing.

Listing Control Pseudo-Ops

There are five pseudo-ops that alter the assembly listing. Table 4-5 lists these pseudo-ops and describes their functions.

Refer to the individual pseudo-op descriptions in Chapter 7 for more detailed information about these pseudo-ops.

Table 4-5. Assembly Listing Control Pseudo-Ops

Pseudo-Op	Description
.EJEC	Begin a new page in the assembly listing (i.e., generate a form feed character)
.NOCON	Enable or suppress the listing of conditional source lines
.NOLOC	Enable or suppress the listing of source lines that lack location fields
.NOMAC	Enable or suppress the listing of macro expansions
.RDXO	Specify the radix (base) for numeric values in the output listing(s)

Asterisks (**)

You may suppress the listing of a line in the source file by placing two consecutive asterisks (**) anywhere in the line. It is good practice to put the two asterisks at the beginning of a line. For example:

Source Code

```
     LDA 0,0,1
**   LDA 0,0,2
     LDA 0,0,3
```

Assembly Listing

```
01 00000'020400        LDA 0,0,1
02 00002'021400        LDA 0,0,3
```

Note that the location counter in the assembly listing (columns 4 through 8) jumps from 00000 to 00002. The Macroassembler assembled all three source lines but did not list the second statement. The asterisks do not alter the object file, only the assembly listing.

If you place two asterisks on a source line that generates an error, the Macroassembler ignores the listing suppression and reports the error.

You may override the ** listing suppression indicator at assembly time by using the /O switch on the MASM command line (see Chapter 8).

Cross-Reference Listing

By default, the Macroassembler generates a cross-reference listing of symbols with every assembly listing. The cross-reference listing provides an alphabetic list of symbols and their values. It also shows the page and line numbers of the assembly listing in which the symbols appear.

For example, suppose the symbol SUB2 has the value 61 8 and appears on the first page, fourth line of your program. The cross-reference shows the symbol SUB2, followed by the value 000061, then the page/line indicator, 1/04.

In addition to that information, the cross-reference listing also identifies the page and line on which you defined (or redefined) the symbol (if applicable). The Macroassembler signals the defining location(s) by placing a number sign (#) after the appropriate page/line indicator.

The cross-reference listing includes several assignment mnemonics that provide additional information about the symbols in your program. Table 4-6 lists the assignment mnemonics and their meanings.

Table 4-6. Cross-Reference Assignment Mnemonics

Mnemonic	Meaning	Defining Pseudo-Op
EN	Entry symbol	.ENT
EO	Overlay entry	.ENTO
MC	Macro symbol	.MACRO
NC	Named common symbol	.COMM
PN	Procedure entry	.PENT
XD	External displacement symbol	.EXTD
XN	External normal symbol	.EXTN
(spaces)	All other symbols	

These mnemonics appear in the cross-reference listing immediately after the symbol's value (where applicable). Figure 4-2 is a sample cross-reference listing.

```
   0003 TOFIX

 AT    000042'       1/53#   1/56
 BLANK 000114'       1/28    2/22    2/38#
 C57   000116'       1/43    2/40#
 COUNT 000113'       1/14    1/24    1/35    2/14    2/37#
 DONE  000072'       2/06    2/18#
 ERROR 000102'       1/37    1/55    2/12    2/26#
 FLAG  000032'       1/33    1/39#
 LOOP  000037'       1/50#   2/15
 MINUS 000115'       1/31    2/39#
 NEXT  000012'       1/23#   1/30
 RADIX 000112'       1/16    2/09    2/36#
 RETUR 000103'       2/27#   2/34
 TEST  000054'       1/58    2/03#
 TOFIX 000000' EN    1/08    1/10#
 UPPER 000056'       1/45    2/05#
 ZERO  000107'       1/26    2/32#
```

Figure 4-2. Sample Cross-Reference Listing

Notice that the default cross-reference listing does not contain references for instruction symbols. You can use the /P switch to add instruction symbols to the listing. This switch is described in Chapter 8.

Error Listing

The *error listing* contains the title of your source module and lists each source line that has an error. The Macroassembler can report up to three errors for the same source statement. The error listing is useful in programs that produce very long listings since it lists only the lines with errors. However, it contains no information that is not in the assembly listing.

The following example shows a sample of source code and the corresponding error listing. The Macroassembler generated the error listing while assembling source file MOD1.SR. MOD1.SR contains the following source code. We have put comments on the source lines that cause errors.

```
        .TITL   MOD1
        .NREL
        LDA 0,C50
        LDA 4,C60
        ADD 0,2
        STA 0,BUFF      ;VALUE OF BUFF IS NOT DEFINED

        MOV 1,1,SNR
        ELDA 0,SYMB     ;SKIP CODED BEFORE 2-WORD INSTRUCTION

DATA:   277777          ;VALUE IS OUT OF RANGE
SYMB:   17
C50:    50
C60:    60
        .END
```

The error listing for this source code is shown below.

```
                      .TITL   MOD1
O00001'020411         LDA 4,C60       ;IMPROPER VALUE (4) IN AC FIELD
U00003'040000         STA 0,BUFF      ;VALUE OF BUFF IS NOT DEFINED
Q00005'122470         ELDA 0,SYMB     ;SKIP CODED BEFORE 2-WORD INSTRUCTION
N00007'077777 DATA:   277777          ;VALUE IS OUT OF RANGE
```

The error listing shows that the assembler detected four errors. The overflow (O) error (line 1) was produced because the instruction's accumulator field was not between 0 and 3. The undefined (U) error was generated because BUFF is not defined in the program (or externally). The Macroassembler produced the questionable line (Q) error on line five because you should not code a skip instruction ahead of a double-word instruction. The number (N) error on line 7 was produced because 277777 does not fit into a 16-bit word.

Appendix C lists all the assembly error codes that the Macroassembler may return.

Error Listing Control

The Macroassembler always produces an error listing; you cannot suppress it. If you do not include the /E= switch on the MASM command line, the Macroassembler reports all errors to the generic file @OUTPUT. If you use the /E=filename function switch, the Macroassembler sends errors to the specified file.

When you produce an assembly listing (i.e., issue the /L or /L=filename function switch), the Macroassembler reports all errors to both the error listing file and the assembly listing file.

Output Function Switches

Throughout this chapter, we have discussed the MASM command line switches that affect Macroassembler output. Table 4-7 lists useful switch combinations, shows what output the Macroassembler produces in each case, and also indicates where that output resides.

The general form for using these switches in the MASM command line is

```
     { MASM   }
XEQ  {        }<function switch>... sourcefile</S>...<nl>
     { MASM16 }
```

Refer to Chapter 8 for descriptions of the /S argument switch, and the MASM function switches. Chapter 8 also has more information about the Macroassembler command line.

Table 4-7. AOS Macroassembler Output Function Switches

Output Function Switches	Assembly Listing	Cross-Reference Listing	Error Reports	Object File	Permanent Symbol Table
NO SWITCHES			@OUTPUT	<sourcefile>.OB	
/L	@LIST	@LIST	@OUTPUT and @LIST	<sourcefile>.OB	
/L = FILE1	FILE1	FILE1	@OUTPUT and FILE1	<sourcefile>.OB	
/E = FILE2			FILE2		
/B = FILE3			@OUTPUT	FILE3.OB	
/N			@OUTPUT		
/L = FILE1 /E = FILE2	FILE1	FILE1	FILE1 and FILE2	<sourcefile>.OB	
/L = FILE1 /E = FILE2 /N	FILE1	FILE1	FILE1 and FILE2		
/L = FILE1 /E = FILE2 /B = FILE3	FILE1	FILE1	FILE1 and FILE2	FILE3.OB	
/S			@OUTPUT		MASM.PS

End of Chapter

Chapter 5
Macros and Generated Numbers and Symbols

Macros

In many cases, you will use a series of source statements repeatedly in one module. Rather than manually inserting the same code at several places, you may assign the source string a name. Then, each time you wish to insert that source code in your module, simply use the assigned name. The Macroassembler automatically substitutes the corresponding code.

We refer to this programming construct as a macro. By using macros in your module, you can greatly simplify assembly language programming. Incidentally, it is from this programming construct that the Macroassembler derives its name.

The following sections of this chapter describe macros and their uses in detail. Refer to Chapter 3 for a discussion about how the Macroassembler actually processes macros.

Macro Definition

To associate a name with a source string, use the .MACRO pseudo-op. The format for using .MACRO is

```
        .MACRO□macro-name
        macro-definition-string
%
```

where:

macro-name is the name you will use to refer to this particular macro. **Macro-name** must conform to the rules for symbols presented in Chapter 2 (see "Macro Symbols").

macro-definition-string consists of one or more source statements. The assembler substitutes these statements for **macro-name** in your module.

% terminates the macro definition string and is not part of the macro definition.

The following source code defines a simple macro and then uses that macro.

```
    .MACRO    FIVES       ;The name of the macro is FIVES.
    5                     ;The macro definition string consists of
    5                     ;two data entries.
%                         ;End of macro definition string.


    FIVES                 ;When the assembler encounters the macro
                          ;name FIVES, it substitutes the macro
                          ;definition string in your module (in
                          ;this case, two consecutive data entries).
```

Within the macro definition string, two characters have special meanings: underscore (_) and uparrow (^). The underscore (ASCII code 1378) directs the assembler to store the next character without interpreting it. Thus, you usually use the underscore to store characters that have special significance when in a macro definition string. In other words, if you precede the characters %, _, or ^ with an underscore, MASM does not interpret them.

For example, if you want to place a percent sign in a source line, you must precede it with an underscore. If you do not, the assembler interprets % as the end of the macro. Thus, if you want to place the string % MEANS PER 100 in a macro, you must enter _% MEANS PER 100. Also, by using the underscore and percent in this fashion, you may write one macro that creates a second macro at expansion time.

If you place an underscore before a character that the assembler would not interpret anyway (i.e., a character other than %, _, or ^), the assembler ignores it. For example, the assembler interprets

```
        .MACRO      X
        A_B                   ;The assembler removes _
%                             ;from the symbol
```

as equivalent to

```
        .MACRO      X
        AB
%
```

Inside a macro, to use a symbol containing an underscore, include an extra underscore in the symbol. The first underscore directs the assembler to store the second one as part of the symbol. Thus, to store the symbol A_B in a macro, enter A__B.

The second character that has a special meaning inside macros is the uparrow (^) (ASCII 136_8). You use this character when defining a macro that accepts arguments. The following section, "Arguments in Macro Definitions," provides information about macro arguments and the uparrow character.

The assembler returns all characters in the macro definition string, except the underscore (_), uparrow (^), and percent (%), exactly as you enter them. The assembler does not automatically insert statement terminator (end-of-line) characters in macro definitions. Thus, you must explicitly terminate each line in your macro definition string with a statement terminator (carriage return, form feed, or NEW LINE).

If you include an expression in your macro definition string, be sure that it appears on one source line. You may not break up an expression with comments or statement terminators. Each expression must be less than or equal to 132 characters in length, the line limit of the assembler.

The % terminator must be the last character in a macro definition. We recommend that you make the terminator the first character on the definition's last line. Both of the following examples use the terminator legally.

```
        .MACRO TEST1
        LDA 0,0,3
        MOV 0,0,SNR
%                     ;last line.

       .MACRO EXP      ;This example shows the terminator
       TEST ^1+^2  %   ;on the same line as the last
                       ;expression in the definition.
```

An MASM macro definition can be suspended and continued later. This feature is useful if one macro is used to define another macro. The first macro may terminate definition of the inner macro temporarily, assign new values, and continue.

The Macroassembler appends the second and subsequent definitions to macros that are already defined. For example:

```
        .MACRO TEST2
        H=2
        I=0
%


        J=0
        K=1
%
```

is the same as:

```
        .MACRO TEST2
        H=2
        I=0
        J=0
        K=1
%
```

To delete a macro definition, use the .XPNG pseudo-op. .XPNG removes all macro names and their definition strings. The .XPNG pseudo-op is described in Chapter 7.

Arguments in Macro Definitions

You may include formal (dummy) arguments in the macro definition string. When you call the macro, you supply an actual value for each formal argument. At expansion time, the assembler replaces the formal arguments in the macro definition string with the actual arguments in the macro call.

This section describes how to place formal arguments in the macro definition string. "Macro Calls" explains how to pass actual arguments to the macro.

Within the macro definition string, all formal (dummy) arguments begin with an uparrow (^) (ASCII 136_8). There are three formats for formal arguments:

^n	where n is a digit from 1 to 9
^a	where a is a letter from A to Z
^?a	where a is a single character from the following set: A - Z, a - z, 0 - 9, ?

A digit following ^ represents the position of an actual argument in the macro call's argument list. That is, when the assembler expands the ADD macro, it replaces all occurrences of ^n with the nth actual argument in the macro call.

For example, in the following macro, the formal argument ^2 appears in the macro definition string. When you call the macro, the assembler replaces ^2 with the second argument in the macro call.

```
        .MACRO      TWO       ;Define macro TWO.
        A=^2                  ;A equals the second argument you
                              ;pass to macro TWO.
%                             ;Macro terminator.


        TWO         3,4       ;Call macro TWO with two arguments.
                              ;The assembler substitutes the second
                              ;argument for ^2. Therefore, A
                              ;now equals 4.
```

The ^n format allows you to reference only the first nine arguments to the macro (^1, ^2,..., ^9). Since the assembler allows you to supply up to 63_{10} arguments, you must use the ^a and ^?a formats to represent arguments 10 through 63_{10}.

The a or ?a following ^ is a user symbol whose value the assembler looks up when expanding the macro. The value of the symbol indicates the position of the actual macro argument that replaces it (as in ^n). The value for a or ?a must be in the range 1-63_{10}, since no macro can have more than 63 arguments.

The following example illustrates the use of ^a and ^?a within a macro definition string.

```
        D=1                     ;Initialize symbols D and ?N.
        ?N=3


        .MACRO      ADD         ;Define macro ADD.
        X=^D+^?N                ;X is the sum of two arguments.
%                               ;Macro terminator.


        ADD         2,4,5       ;Call macro ADD with three arguments.
```

When MASM expands the ADD macro, D equals 1 and ?N equals 3. Thus ^D evaluates to ^1 and ^?N evaluates to ^3. Consequently, MASM converts the statement X=^D+^?N to X=^1+^3. After the call to macro ADD, X has the value of the first argument plus the third argument (X=2+5).

A zero or negative number following an uparrow (e.g., ^0, ^-5) is unconditionally replaced by the null string (a string with no characters). Similarly, MASM substitutes the null string for any formal argument value that is larger than .ARGCT (.ARGCT equals the number of actual arguments you supply to the macro call). For example, MASM substitutes the null string for ^3 if you supply only two arguments when calling the macro. These rules apply to all three formal argument formats (i.e., ^n, ^a, and ^?a).

Macro Calls

After defining a macro, you can issue the macro name wherever you want to insert the macro definition string in your module. We refer to the source line that calls the macro as a macro call.

A macro call consists of the macro name defined in the .MACRO statement followed by actual arguments to replace any formal arguments in the macro definition string. You may call a macro any number of times in your source module.

Calling Macros without Arguments

If your macro definition string contains no formal arguments, simply enter the macro name on a source line. The Macroassembler inserts the corresponding macro definition string in your object module.

Thus, the syntax for calling a macro without arguments is

macro-name

where:

macro-name is the name you assigned to a macro definition string in a .MACRO statement

The following example shows this type of macro call:

```
        .MACRO    FOURS         ;Define macro FOURS
        4                       ;(no formal arguments).
        4
%

        FOURS                   ;Call to macro FOURS (no arguments).
```

Calling Macros with Arguments

If your macro definition string contains formal arguments, you must supply actual arguments in the corresponding macro call(s). There are two formats for passing arguments to macros:

1. **macro-name□arg...**

2. **macro-name□[arg...**<nl>
   arg...]

where:

macro-name is the name you assigned to a macro definition string in a .MACRO statement

arg is an actual argument that you pass to macro **macro-name**

During macro expansion, the assembler replaces formal arguments in the macro definition string with the actual arguments in the macro call. If you supply more than one argument, separate them with spaces, horizontal tabs, and/or one comma.

In most cases, you use the first macro call format. Simply enter the macro name followed by the actual arguments on the same line. The following example illustrates this form of macro call:

```
        .MACRO    FORM1       ;Define macro FORM1.
        ^1                    ;The macro definition string
        LDA       ^2,^3       ;contains 3 formal arguments.
%


        FORM1     5,2,DATA    ;Call macro FORM1 with 3 arguments.
```

The assembler substitutes 5, 2, and DATA for ^1, ^2, and ^3, respectively. Thus, the FORM1 macro call generates the following two source lines:

```
        5
        LDA             2,DATA
```

If the arguments in your macro call extend to a second source line, you must enclose them in square brackets; that is, use macro call form 2.
The following example shows this form of macro call:

```
        .MACRO    FORM2       ;Define macro FORM2.
        ^1                    ;The macro definition contains
  ^2:   LDA       ^3,^4       ;4 formal arguments.
%


        FORM2     [5, START,
1, DATA]                      ;Macro call to FORM2.
```

The assembler substitutes 5, START, 1, and DATA for ^1, ^2, ^3, and ^4, respectively. Thus, the FORM2 macro call generates two source lines:

```
        5
 START: LDA             1,DATA
```

Note that MASM processes both forms of macro calls in the same manner. That is, the call form you use does not influence the macro expansion.

The actual arguments you pass to a macro may be integers, symbols, or expressions. However, you must be sure that the value of an actual argument is legal for the corresponding field in the macro definition string. For example:

```
     .MACRO    LOAD         ;Macro name is LOAD.
     LDA       ^1,DATA      ;The first argument to LOAD goes
%                           ;in the AC field of the LDA instruction


     A=6
     LOAD      A            ;Call macro LOAD with an argument.
```

This macro call causes an error because it generates the instruction LDA 6,DATA. The value 6 is too wide for the two-bit accumulator (AC) field.

```
        A=2
        LOAD    A          ;Call macro LOAD with an argument.
```

This macro call is acceptable because LDA 2,DATA is a legal instruction.

If you supply more actual arguments in the macro call than formal arguments in the macro definition, the assembler ignores the excess arguments. That is, the assembler ignores all arguments in the macro call that do not have counterparts in the macro definition string.

If you do not supply enough actual arguments in your macro call, the assembler substitutes null strings (strings with no characters) for excess formal (dummy) arguments. For example, if you include formal argument ^3 in your macro definition string but only supply two arguments when you call the macro, MASM replaces ^3 with the null string.

No macro call can have more than 63_{10} arguments.

Passing Special Characters and Null Arguments to Macros

The previous discussion showed how to use square brackets to extend a macro call onto a second source line. You need to use square brackets in macro calls in two other situations:

- when passing special characters as arguments to a macro
- when passing null arguments to a macro

Special Characters

You must enclose certain characters in square brackets if you intend to pass them as arguments to a macro. These characters are

```
@       #       **      =       :       ;       \
```

For example, suppose you want to pass the semicolon character (;) as an argument to macro MACRO1. You would issue the source statement

```
        MACRO1          [;]
```

You could not simply say

```
        MACRO1          ;
```

because the Macroassembler would interpret the semicolon as the beginning of a comment string, not as an argument to the macro. That is, the second statement calls MACRO1 with no arguments.

To pass a special character to a macro along with other arguments, you may place either all the arguments or only the special character inside square brackets. Thus, the following two macro calls are equivalent:

```
        MACRO2          [;],4


        MACRO2          [; , 4]
```

Both of these statements call macro MACRO2 with the two arguments ; and 4.

You may never pass certain characters as arguments in a macro call (even within square brackets). These characters are the carriage return, form feed, and NEW LINE.

Null Arguments

In certain situations, you may wish to pass null arguments to a macro. A null argument is a string with no characters (a string of length zero).

To pass a null argument to a macro, simply enter square brackets that enclose no characters, []. The Macroassembler substitutes the null string for the corresponding argument in the macro definition string.

The following example defines a macro containing three formal arguments:

```
        .MACRO          ADDR
         LDA            0,^1^2,^3
%
```

When you call this macro, pass a displacement value and an addressing index in the second and third arguments, respectively. In the first argument, you may indicate indirect addressing by supplying the character @, or you may pass a null string.

Our first call to macro ADDR passes @.

```
        ADDR            [@],LOC1,2
```

This macro call generates the source statement LDA 0,@LOC1,2. Note that we passed the @ character inside square brackets (see the previous discussion on special characters).

If you want to call macro ADDR without indicating indirect addressing, you must pass a null string in the first argument. That is, the macro call

```
        ADDR            [],LOC1,2
```

generates the source statement LDA 0,LOC1,2. Again, the Macroassembler substitutes the null string for ^1 in macro ADDR.

You should note that two consecutive commas in a macro call also indicate a null argument. For example, the macro call

```
        MACRO3          0,,2
```

calls macro MACRO3 with three arguments; the second argument is null.

In general, we recommend that you use square brackets, not consecutive commas, to indicate null arguments since they improve your program's readability.

Macro Expansions in Assembly Listings

When you issue a macro call, MASM substitutes the assembled macro definition string in the binary object file. However, the assembly listing shows both the macro call and the macro expansion. Figure 5-1 illustrates a macro in source code and the corresponding listing. Note that the assembly listing contains both the macro call DSP 2 and the macro's expansion.

```
            Source Text

        .MACRO          DSP     ;MACRO DEFINITION
        ^1
%

        DSP 2                   ;CALL TO MACRO DSP

           Assembly Listing

01                      .MACRO DSP      ;MACRO DEFINITION
02                      ^1
03              %
04                      DSP 2           ;CALL TO MACRO DSP
04 00000'000002         2               ;WITH ARGUMENT
```

Figure 5-1. Macro Listing

The assembly listing values for the location counter (columns 4-8) and the data field (columns 10-15) show that the binary object file contains only the macro expansion. That is, although the listing says

```
        DSP     2
        2
```

the object file contains the binary code for

```
        2
```

You may suppress the listing of macro expansions by using the .NOMAC pseudo-op. If you suppress expansions, the assembly listing shows only the macro call. The .NOMAC pseudo-op does not affect the object file in any way.

The following example shows the result of suppressing macro expansions.

```
                     .MACRO Z        ;DEFINE MACRO Z.
                     5
                     LDA ^1,^2
                     %
                                     ;MACRO EXPANSIONS ARE
                     Z 0,4           ;LISTED BY DEFAULT.
00000'000005         5
00001'020004         LDA 0,4

      000001         .NOMAC 1        ;DIRECT THE ASSEMBLER TO
                                     ;SUPPRESS LISTING OF
                                     ;MACRO EXPANSIONS (I.E., PASS
                                     ;A NONZERO VALUE TO .NOMAC).
                     Z 0,4
      000000         .NOMAC 0        ;REENABLE THE LISTING OF MACRO
                                     ;EXPANSIONS (I.E., PASS A ZERO
                                     ;VALUE TO .NOMAC).
                     Z 0,4
00004'000005         5
00005'020004         LDA 0,4
```

Again, the .NOMAC setting does not affect macro expansions in the object file. The assembler expands all macro calls correctly, although it may not list those expansions. This explains why the location counter jumps from 1 to 4 in the above listing example; the missing locations represent macro expansions whose listings were suppressed.

You may override the .NOMAC pseudo-op at assembly time by using the /O switch in the MASM command line. Chapter 8 provides more information about this switch.

The action performed by the two asterisks ** (the no-listing indicator) is unique in macro calls that extend to more than one line. If the first line of the macro call starts with two asterisks, the assembler does not print the last line of arguments. MASM will, however, assemble the macro correctly.

Macro-Related Pseudo-Ops

In addition to .MACRO and .NOMAC (described above), the assembler provides two other pseudo-ops you may use with macros: .ARGCT and .MCALL.

The .ARGCT pseudo-op is a value symbol that returns the number of actual arguments you pass to a macro. Use this symbol inside the macro definition string. For example:

```
01                          .MACRO X
02                          ^1+^2
03                          (.ARGCT)
04                          %
05
06                          X 4,5    ;PASS TWO ARGUMENTS TO X
07 00000'000011             4+5
08 00001'000002             (.ARGCT)

        ;AT EXPANSION TIME, THE VALUE FOR .ARGCT IS 2
        ;BECAUSE MACRO X WAS CALLED WITH TWO ARGUMENTS
```

The .MCALL pseudo-op is also a value symbol that you may use inside a macro definition string. This symbol has the value 0 if this is the first call to that macro on this assembler pass. The symbol has a value of 1 if this is not the first call in the current pass. For example:

```
    .MACRO      Y
    .IFE        .MCALL        ;Assemble all code up to .ENDC only
    JSR         @FIRST        ;if the value of .MCALL equals zero.
    .ENDC
%
```

The first time you call macro Y, .MCALL equals 0. Thus, the .IFE condition will be true, and MASM will assemble the statement in the conditional block (JSR @FIRST). However, on subsequent calls to macro Y, .MCALL will equal 1 and MASM will not assemble the .IFE block.

Chapter 7 provides more detailed descriptions of .ARGCT, .MACRO, .MCALL, and .NOCON.

Loops and Conditionals in Macros

When you use a .DO loop or an .IF conditional inside a macro, be sure you include a corresponding .ENDC pseudo-op in that same macro. MASM takes one of the following actions if it encounters a macro definition terminator % before an .ENDC:

- * MASM ignores a .DO statement inside a macro if there is no corresponding .ENDC.
- * If you do not terminate an .IF conditional inside a macro, MASM ends the conditional immediately before the macro definition terminator %.

The remainder of this section shows the correct and incorrect use of .DO loops and .IF conditionals inside macros.

The following example shows the proper use of a .DO loop:

```
    .MACRO      LOOP
    .DO         ^1      ;When you call this macro, the first
        3               ;argument indicates how many times to
        4               ;assemble the .DO loop.
    .ENDC               ;The end of the .DO loop is inside the
%                       ;macro definition string.
```

Next is an example of an incorrect .DO loop:

```
    .MACRO          ERRDO
    .DO             5          ;Incorrect use of .DO in a macro
    6                          ;(no .ENDC pseudo-op).
%
```

When you call macro ERRDO, the Macroassembler ignores the .DO statement since it is not terminated inside the macro (i.e., no corresponding .ENDC). The Macroassembler will, however, assemble the data entry 6 correctly, but only once.

The following code shows the correct use of an .IF pseudo-op inside a macro:

```
.MACRO          COND        ;If the first argument you pass to this
.IFE            ^1          ;macro equals 0, MASM assembles the data
        10                  ;entries 10 and 20.  Note that the .ENDC
        20                  ;statement appears inside the macro.
.ENDC
30                          ;MASM assembles data entries 30 and 40
40                          ;regardless of the argument value you pass
                            ;to this macro.
%
```

The following example shows an improper .IF conditional inside a macro:

```
    .MACRO          ERRIF
    .IFE            ^1         ;Improper use of IFE in a macro.
      50                       ;(no .ENDC pseudo-op).
      60
%
```

In this case, the .IFE has no corresponding .ENDC within the macro definition. Therefore, MASM assumes the conditional code ends immediately before the % statement.

Macro Examples

The following examples illustrate the use of the MASM macro facility. Refer to Chapter 7 for descriptions of any pseudo-ops that you are not familiar with.

Example 1: Logical OR

Our first example is a macro that computes the logical OR of two accumulators. Of course, you could use the IOR instruction to perform this operation; we present the example only to illustrate the macro facility.

```
       .MACRO          OR
        COM            ^1,^1          ;Complement AC^1.
        AND            ^1,^2          ;Clear ON bits of AC^1.
        ADC            ^1,^2          ;OR result to AC^2.
%
```

The call format for macro OR is similar to an ALC instruction.

OR□acs□acd

where:

OR is the name of the macro

acs is the source accumulator

acd is the destination accumulator

The following macro call shows how the Macroassembler expands macro OR. Note that MASM substitutes arguments in the comments as well as the instructions.

```
06                     OR 1,2
07 00000'124000        COM        1,1        ;COMPLEMENT AC1.
08 00001'133400        AND        1,2        ;CLEAR ON BITS OF AC1.
09 00002'132000        ADC        1,2        ;OR RESULT TO AC2.
```

Example 2: Factorial

Our second example illustrates the recursive property of macros. Macro FACT computes factorials of positive integers. Its input consists of an integer I and a variable V, and it computes the value

```
V = I!
```

using the recursive formula

```
I! = I*(I-1)!
```

The macro definition for FACT is

```
        .NOCON    1
        .MACRO          FACT
**      .DO             ^1>1
                FACT    ^1-1,^2
                ^2=^1*^2
**      .ENDC
**      .DO             ^1<=1
                ^2=1
**      .ENDC
**      .DO             ^1<0
                ^2=0
**      .ENDC
%
```

When you issue the statement

```
      FACT□I□V
```

MASM computes the factorial of integer I and stores the result in variable V.

MASM expands macro FACT as follows:

* If I is greater than 1, macro FACT calls itself recursively with successively smaller values for I. As these levels expand to completion, MASM computes I! and stores the result in variable V.

* If I is 0 or 1, FACT returns the value 1 in variable V. The second .DO conditional performs this test.

* If I is negative, FACT returns the value of 0 in variable V. The last .DO conditional performs this test.

The following call to FACT computes the factorial of 4 and stores the result in variable A. The macro listing shows only the recursive calls to FACT and the subsequent computation statements; the ** indicators suppress the listing of all other macro statements.

```
15                FACT 4,A
16                                FACT    4-1,A
17                                FACT    4-1-1,A
18                                FACT    4-1-1-1,A
19       000001                   A=1
20       000002                   A=4-1-1*A
21       000006                   A=4-1*A
22       000030                   A=4*A
```

Example 3: Packed Decimal

This example macro stores numeric values in "packed decimal" format. In packed decimal, each decimal digit requires 4 bits for its representation. Thus, a byte can contain two packed decimal digits, and a 16-bit word can hold four digits.

Our macro outputs the least significant word of the packed decimal representation first. The number's sign occupies the least significant (rightmost) 4 bits of the first word.

The translation from decimal to 4-bit binary is

Decimal	4-Bit Binary
+	0011 (same bit pattern as "3")
-	0100 (same bit pattern as "4")
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The input to macro PACK consists of a string of decimal digits separated by delimiters and followed by an explicit sign (+ or -) and the precision in 16-bit words.

PACK□d<□d>...□s□w

where:

PACK is the name of the macro.

d is a decimal digit. If you supply more than one digit, separate them with spaces, tabs, or commas.

s is the sign of the digit (+ or -).

w is the precision and indicates how many 16-bit words MASM will allocate for the packed decimal representation.

Within macro PACK, the input radix must be decimal. So, PACK saves the initial input radix and changes it to decimal for the macro expansion. Before returning, PACK restores the original input radix.

To present the output in 4-bit quantities, the output radix must be hexadecimal (base 16). Again, PACK saves the initial value at the beginning of the macro and restores it at the end.

Though MASM assembles many statements for each macro call, the listing shows only the assembled storage words that hold the packed decimal value.

The macro definition for PACK follows:

```
          .MACRO          PACK
**        .PUSH           .NOMAC
**        .NOMAC          1

          .PUSH           .RDX
          .PUSH           .RDXO
          .RDX            10
          .RDXO           16

          I=.ARGCT
          J=I-1
          B=11
          W=3+(("^J)-("+)/2)
          J=J-1

          .LOC            .+^I-1
          .DO             ^I
                 .DO                B+1/4
                          W=W+0^JBB
                          B=B-4
                          .DO                 J<>0
                                   J=J-1
                          .ENDC
```

```
                       .ENDC
**                     .NOMAC  0
                       W
**                     .NOMAC  1
                       W=0
                       B=15
                       .LOC    .-2
                .ENDC

                .LOC                .+^I+1
                .RDXO               .POP
                .RDX                .POP
**              .NOMAC              .POP
%
```

The following listing shows four calls to macro PACK and the corresponding expansions:

```
38        000100         .LOC    100      ;START AT ADDRESS 100
39                                        ;OR 40 (HEXADECIMAL)
40
41                       PACK 1 2 +,2
42 00041    0123                W
43 00040    0000                W
44
45                       PACK 1 2 3 4 5 +,3
46 00044    3453                W
47 00043    0012                W
48 00042    0000                W
49
50                       PACK 1 2 3 4 5 -,3
51 00047    3454                W
52 00046    0012                W
53 00045    0000                W
54
55                       PACK 9 8 7 6 5 +,6
56 0004D    7653                W
57 0004C    0098                W
58 0004B    0000                W
59 0004A    0000                W
60 00049    0000                W
01 00048    0000                W
```

System Calls

The AOS software package includes a number of system calls. System calls are actually assembly language macros that perform common operations (for example, record I/O). The advantage to using system calls is that you do not have to code the macros yourself; you simply issue the system call.

The AOS Programmer's Manual documents all system calls.

Generated Labels

The dollar sign ($) can be used to generate unique labels within macros. In non-string mode, each occurrence of the character $ is replaced by three characters from the set 0-9 and A-Z. The Macroassembler determines which three characters replace the dollar sign by converting a count of the number of macro calls in radix 36 to ASCII. In nested macros, the replacement string for $ in the outer macro is saved and restored when the inner macro has been expanded.

When used in labels, $ should generally not be the first character on a line. The first replacement character could be a digit. This would generate an error, because a symbol cannot start with a digit.

The following example shows a macro that generates labels. The listing shows how a new label is generated each time the macro is called.

```
        .MACRO  BKT     ;Macro to generate labels
        DSZ     COUNT
TR$=    .               ;Label entry generated by macro
%
COUNT:  .
06       000005         .DO 5           ;CALL THE MACRO FIVE TIMES
07                      BKT
08 00001'014777         DSZ     COUNT
09       000002'TR$001= .               ;THIS LABEL GENERATED BY MACRO
10                      BKT
11 00002'014776         DSZ     COUNT
12       000003'TR$002= .               ;THIS LABEL GENERATED BY MACRO
13                      BKT
14 00003'014775         DSZ     COUNT
15       000004'TR$003= .               ;THIS LABEL GENERATED BY MACRO
16                      BKT
17 00004'014774         DSZ     COUNT
18       000005'TR$004= .               ;THIS LABEL GENERATED BY MACRO
19                      BKT
20 00005'014773         DSZ     COUNT
21       000006'TR$005= .               ;THIS LABEL GENERATED BY MACRO
```

Generated Numbers and Symbols

You may direct the assembler to generate numbers and symbols by using the following format:

\symbol

At assembly time, MASM replaces **\symbol** with a 3-digit number representing the value of **symbol.** The assembler uses the current input radix for this substitution and truncates the value of **symbol** to three characters, if necessary.

A **\symbol** may stand alone in code to form an integer. It may also follow characters that, together with the value of **\symbol,** will form a number or symbol. For example,

```
          A=2            ;Initialize A and B.
          B=1234
X\A:      1              ;X\A evaluates to the symbol X002.
X\B:      1              ;X\B evaluates to the symbol X234
                         ;(the "1" is truncated from "1234").
          C=\A+\B        ;\A equals 002 and \B equals 234 so
                         ;C equals 236.
          450.\A         ;450.\A evaluates to 450.002
```

A symbol must be generated with exactly one **\symbol** string. For example, the string A\I\J will cause an undefined symbol (U) error.

The assembly listing for a generated number or symbol shows both the replacement value and the **\symbol** designation. For example, the above section of source code would appear as follows in the assembly listing:

```
01       000002            A=2
02       001234            B=1234
03 00000'000001 X\A002:       1
04 00001'000001 X\B234:       1
05
06       000236            C=\A002+\B234
07
08 00002'041434            450.\A002
09       020010
```

Table 5-1 shows the correspondence between source code, the assembly listing, and the cross-reference listing. Notice that the **\symbol** string (\ONES) is shown in the assembly listing, even though it is suppressed in the cross reference.

Table 5-1. Generated Symbols in Source and Listings

Source Code	**Assembly Listing**	**Cross-Reference Listing**
ONES=111	ONES=111	ONES
A\ONES	A\ONES111	A111

You may increment **\symbol** just as you would increment any other value. For example, the following code creates labels for a table.

```
           .RDX      8
**         X=0                 ;Initialize counter X (** means
**                             ;suppress listing of this line).
TABLE:     .DO       64.       ;Assemble this loop 64 (decimal) times.
A\X:                 0         ;Create labels A000, A001,...,A077.
**                   X=X+1     ;Increment counter X.
**         .ENDC               ;End of .DO loop.
```

The listing for this section of code follows:

```
01         000010              .RDX      8
03         000100 TABLE:       .DO       64.
04 00000'000000 A\X000:                     0
05 00001'000000 A\X001:                     0
06 00002'000000 A\X002:                     0
07 00003'000000 A\X003:                     0
03 00073'000000 A\X073:                     0
                                .
                                .
                                .
06 00076'000000 A\X076:                     0
07 00077'000000 A\X077:                     0
```

End of Chapter

Chapter 6
Types of Pseudo-Ops

Pseudo-ops direct the execution of the Macroassembler and represent internal assembler variables. All pseudo-ops fall into one of the following categories:

- Location counter and memory management
- File termination
- Repetitive and conditional assembly
- Macros
- Data placement
- Intermodule communication
- Listing control
- Stack control
- Radix control
- Text strings
- Symbol table
- Symbol deletion
- Miscellaneous

The following sections of this chapter describe each of these categories and list the corresponding pseudo-ops. Each section has a table that indicates whether a pseudo-op may direct the assembly process (*an assembler directive*) and/or represents an internal assembler variable (*a value symbol*).

Refer to "Pseudo-Ops" and "Permanent Symbols" in Chapter 2 for a general description of pseudo-ops. Chapter 7 describes the individual pseudo-ops in alphabetical order.

Location Counter and Memory Management

Generally, you use location counter and memory management pseudo-ops to assign memory locations to the code and data in your source module. More specifically, the pseudo-ops in this category allow you to

- reserve a block of storage locations;
- set the location counter to a specific value; and
- place source code in predefined memory partitions.

Table 6-1 lists the location counter and memory management pseudo-ops.

Table 6-1. Location Counter and Memory Management Pseudo-Ops

Pseudo-Ops	Assembler Directive	Value Symbol	Description
. (period)	NO	YES	Return the value of the current location counter
.BLK	YES	NO	Reserve a block of memory locations
.LOC	YES	YES	Set the current location counter
.LPOOL	YES	NO	Dump the currently defined literals into a data block (must be used with the .NLIT pseudo-op)
.NLIT	YES	NO	Assign literals to NREL instead of ZREL
.NREL	YES	NO	Specify a shared or unshared normal relocatable memory partition
.ZREL	YES	NO	Specify the lower page zero relocatable memory partition

The single character pseudo-op . (period) is a value symbol that represents the current location counter. The location counter is an assembler variable that holds the address and relocation base of the next memory location the Macroassembler will assign.

The .ZREL and .NREL pseudo-ops direct the assembler to place the source lines that follow them into predefined memory partitions. Use .ZREL for relocatable data that must reside in lower page zero (locations 50-377 8). Use .NREL for relocatable data that can be anywhere else in memory (locations 400 8 to 32KW-1). The .NREL pseudo-op allows you to place source code in either shared or unshared partitions.

Use the .LOC pseudo-op to set the location counter to a specific value within a memory partition.

Refer to Chapter 3 for a detailed discussion of how the assembler assigns addresses to the words in your source module. That chapter also describes the various memory partitions available for your use.

File Termination

The assembler has two file termination pseudo-ops (see Table 6-2).

Table 6-2. File Termination Pseudo-Ops

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.END	YES	NO	End-of-program indicator
.EOF	YES	NO	Explicit end-of-file

The .END pseudo-op terminates the source code you pass to the assembler. Also, by passing an address to .END, you may indicate where you want a program to begin at execution time.

Use .EOF when you include more than one source file on the MASM command line. When you place this pseudo-op at the end of a file, you inform the assembler that the current input file is complete but more files may follow. All files, except the last one, should end with the .EOF pseudo-op; the last one should end with .END.

Repetitive and Conditional Assembly

The pseudo-ops in this category allow you to

* assemble a series of source lines a specified number, or

* conditionally assemble or by-pass source lines based on the evaluation of an expression (conditional assembly).

Table 6-3 lists the pseudo-ops you use to perform the above operations.

Table 6-3. Repetitive Assembly and Conditional Pseudo-Ops

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.DO	YES	NO	Assemble the following source lines a specified number of times
.ENDC	YES	NO	Define the end of repetitive or conditional assembly lines
.IFE	YES	NO	Assemble the following source lines only if the value of the supplied expression equals zero
.IFG	YES	NO	Assemble the following source lines only if the value of the supplied expression exceeds zero
.IFL	YES	NO	Assemble the following source lines only if the value of the supplied expression is less than zero
.IFN	YES	NO	Assemble the following source lines only if the value of the supplied expression does not equal zero
.GOTO	YES	NO	Suppress assembly of source lines until the specified symbol is encountered

Use the .DO pseudo-op to implement a loop at assembly time. This allows you to assemble a section of source code more than once.

The .IF pseudo-ops assemble a section of code only if an expression satisfies a certain condition. For example, the .IFE pseudo-op directs the assembler to process a section of code only if the argument to .IFE equals zero.

You must use .ENDC to terminate source lines that you want assembled repetitively (with .DO) or conditionally (with .IFE, .IFG, .IFL, or .IFN).

Refer to Chapter 5 for information about how to use repetitive assembly and conditional pseudo-ops inside macros.

Macros

The Macroassembler provides three pseudo-ops that you may use with macros. Table 6-4 describes each of these symbols.

Table 6-4. Macro-Related Pseudo-Ops

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.ARGCT	NO	YES	Return the number of arguments passed to a macro
.MACRO	YES	NO	Defines the start of a macro, and names the macro
.MCALL	NO	YES	Indicate whether a macro has been called on the current assembler pass

Chapter 5 provides a complete description of macros and discusses the use of the above pseudo-ops.

Intermodule Communication

Intermodule communication pseudo-ops allow you to define symbols and data in one source module and to reference that information from a separately assembled module. These pseudo-ops declare entry points, external symbols, and both labeled and unlabeled common areas. Table 6-5 lists the intermodule communication pseudo-ops.

Table 6-5. Intermodule Communication Pseudo-Ops

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.ASYM	YES	NO	Define an accumulating symbol
.COMM	YES	NO	Reserve a labeled common area for intermodule communication
.CSIZ	YES	NO	Reserve an unlabeled common area for intermodule communication
.ENT	YES	NO	Define one or more external entries
.ENTO	YES	NO	Define an overlay entry
.EXTD	YES	NO	Define one or more external displacement references (external symbol value is 8 bits or less)
.EXTN	YES	NO	Define one or more external normal references (external symbol value is 16 bits or less)
.EXTU	YES	NO	Treat undefined symbols as external displacements
.GADD	YES	NO	Assign an expression value to a symbol
.GLOC	YES	NO	Initialize data fields relative to an external symbol
.GREF	YES	NO	Assign an expression value to a symbol without affecting the sign bit
.PENT	YES	NO	Define a procedure entry
.PTARG	YES	NO	Generate a procedure description for ?RCALL, ?KCALL, or ?RCHAIN

.ASYM defines an accumulating symbol, which you then set to an initial value (i.e., 1). In subsequent modules, the symbol is redefined as an accumulating symbol, and is set to a new value (i.e., 2). References to the symbol in succeeding modules will be resolved as the sum (accumulation) of all its previous values (i.e., 3). The symbol retains its relocation type.

The .COMM and .CSIZ pseudo-ops reserve common areas for intermodule communication. Using these pseudo-ops, you may create data storage areas that are accessible to each module in your program. Use .COMM when you want to assign a name to the common area (labeled common area); use .CSIZ to create an unlabeled common area.

An .ENT symbol may represent either an address or a data value that is available for use by separately assembled modules. The separately assembled modules that reference the .ENT symbol must each declare it with an .EXTD or .EXTN pseudo-op; these pseudo-ops inform the assembler that the symbol is defined externally (in a separately assembled source module).

The symbols that the .EXTD and .EXTN pseudo-ops declare differ in the number of bits necessary to represent their values. You may use an .EXTD symbol (external displacement) in any field that is at least 8 bits wide. Use an .EXTN symbol (external normal) in fields where at least 16 contiguous bits are available.

When you declare a symbol as externally defined, the assembler associates the declared field width with that symbol. Each time you use the symbol in your module, MASM verifies that the number of contiguous bits available is at least as large as the symbol's declared width. If you use an externally defined symbol in a field that is not wide enough, the assembler returns an error.

For example, if you issue the statement

```
        .EXTN          A
```

the assembler assumes that the value of A requires a field of at least 16 bits. Each time you use A in this source module, MASM checks that the corresponding field is at least 16 bits wide. If you use A in a field with less than 16 contiguous bits available, MASM returns an error.

The declared field width should correspond to the value of the externally defined symbol. If the declared field width is smaller than the symbol's value, you may receive an error at link time. In the above example, we declared A as a 16-bit value (.EXTN A). If A turns out to have a value larger than 16-bits, Link may not be able to fit A's value in the appropriate fields.

The following example will help clarify the use of the .ENT, .EXTD, and .EXTN pseudo-ops.

Source Module X

```
          .TITL       X      ;Source module X.
     .ENT        H,I,J   ;Symbols H, I, and J are defined in
                         ;this module and may be referenced
                         ;by other modules.
     .ZREL
H:   100                 ;H is a label in lower page zero (i.e.,
                         ;it can be represented with 8 bits).
     .NREL
I:                       ;The values of I and J require 16 bits.
      J=25437
     .END                ;End of source module X.
```

Source Module Y

```
          .TITL  Y       ;Source module Y (assembled separately
                    ;from module X).
     .EXTD  H,I     ;H and I are externally defined and their
                    ;values can be expressed with 8 bits.
     .EXTN  J       ;J is externally defined and its value
                    ;requires a 16-bit field.
     .NREL
     LDA    0,H     ;NO ERROR: The value of H can fit
                    ;into the 8-bit address field provided
                    ;by the LDA instruction.
     LDA    1,J     ;ERROR: The value of J requires 16 bits
                    ;and the LDA address field is only 8
                    ;bits wide.
     EISZ  J        ;NO ERROR: The EISZ instruction
                    ;provides a 15-bit displacement field,
                    ;sufficient for .EXTN symbol J.
     EJMP   H       ;NO ERROR: The instruction EJMP provides
                    ;a 15-bit displacement field.  Since the
                    ;value of H can be represented in 8 bits,
                    ;this statement is legal.

     JMP  H         ;NO ERROR:  The value of H can fit
                    ;into an 8-bit displacement field.
     JMP   J        ;ERROR: The .EXTN statement defines J as
                    ;a 16-bit value.  Since this value cannot
                    ;fit into the 8-bit field that JMP
                    ;provides, MASM returns an error
                    ;for this statement.
     LDA    0,I     ;LINK ERROR: The .EXTD statement defines I
                    ;as an 8-bit value and the LDA instruction
                    ;provides an 8-bit address field.  Thus, MASM
                    ;does not return an error. However, the actual
                    ;value of I, known at link time, will not fit
                    ;into the 8-bit LDA address field so Link will
                    ;report an error.
     .END
```

The .PENT pseudo-op names procedure entries that will gain control when called by one of the following operating system calls: ?RCALL, ?KCALL, or ?RCHAIN.

The .PTARG pseudo-op accepts the name of an external general procedure entry point and generates a procedure descriptor. This descriptor can be passed on the user stack to one of the general procedure calls (?RCALL, ?KCALL, or ?RCHAIN).

The .ENTO pseudo-op is used when a program is to become an overlay within an overlay segment. .ENTO identifies the number and node of an overlay so that it can be called by the ?OVLOD operating system call.

Listing Control

The assembler provides several pseudo-ops that allow you to control the format and content of the assembly listing. Table 6-6 describes the listing control pseudo-ops.

Table 6-6. Listing Control Pseudo-Ops

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.EJEC	YES	NO	Begin a new listing page
.NOCON	YES	YES	Enable or suppress the listing of conditional source lines
.NOLOC	YES	YES	Enable or suppress the listing of source lines that lack location fields
.NOMAC	YES	YES	Enable or suppress the listing of macro expansions

The .EJEC pseudo-op directs MASM to begin a new page in the assembly listing (after listing the .EJEC source line).

Use .NOCON, .NOLOC, and .NOMAC to inhibit (and enable) portions of the assembly listing. Each of these pseudo-ops also functions as a value symbol that returns the last value you supplied to that pseudo-op.

You may generate an assembly listing by including the /L or /L= switch on the MASM command line. By default, the assembly listing contains all source lines in your module and a cross-reference listing of symbols. Use the /O switch to override the .NOCON, .NOLOC, and .NOMAC pseudo-ops at assembly time.

Refer to Chapter 4 for a description of the assembly and cross-reference listings. Chapter 8 explains the MASM command line switches.

Stack Control

The Macroassembler maintains a push-down stack that you may use to save the value and relocation property of any valid assembler expression. In a push-down stack, the last expression you place on the stack is the first you remove.

Table 6-7 lists the stack control pseudo-ops available for your use.

Table 6-7. Stack Control Pseudo-Ops

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.POP	YES	YES	Return the value and relocation property of the last expression pushed onto the stack. .POP removes (pops) this information from the stack.
.PUSH	YES	NO	Push the value and relocation property of an expression onto the stack
.TOP	NO	YES	Returns the value and relocation property of the last expression pushed onto the stack. .TOP does not remove (pop) this information from the stack.

To place a value on the stack, issue the .PUSH pseudo-op. Use the .POP value symbol to access the information on the stack. When you use .POP, the assembler returns the top value (the last one pushed) and removes that entry from the stack. The .TOP pseudo-op returns the value and relocation property of the last expression pushed onto the stack. .TOP does not remove this information from the stack.

Note that the stack we describe above functions at assembly time; it is separate and distinct from the stack that the ECLIPSE computer maintains at execution time.

Radix Control

The radix control pseudo-ops allow you to specify both the input radix and the output radix for your source module. The assembler uses the input radix to interpret numeric expressions in your source code and the output radix to present numeric expressions in the various output listings.

Table 6-8 describes the two radix control pseudo-ops.

Table 6-8. Radix Control Pseudo-Ops

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.RDX	YES	YES	Set radix for numeric input conversion
.RDXO	YES	YES	Set radix for numeric output conversion

Input and output radixes are entirely distinct, and you set them independently. Using the .RDX and .RDXO pseudo-ops, you may specify input and output radixes in the range 8-16. The default radix for both input and output is 8 (octal).

You may use .RDX and .RDXO as value symbols. They return the current input and output radixes, respectively.

Text Strings

The text string pseudo-ops assemble character strings into their equivalent ASCII codes. That is, using these pseudo-ops, you may enter a character string in your source code and have the assembler store its ASCII representation in memory.

Table 6-9 lists the text string pseudo-ops.

Table 6-9. Text String Pseudo-Ops

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.TXT	YES	NO	Store the ASCII value of a text string in consecutive words of memory
.TXTE	YES	NO	Set the leftmost bit for even byte parity
.TXTF	YES	NO	Set the leftmost bit to one unconditionally
.TXTM	YES	YES	Specify left/right or right/left bytepacking within words
.TXTN	YES	YES	Terminate an even-length byte string with no null bytes or two null bytes
.TXTO	YES	NO	Set the leftmost bit for odd byte parity

Using .TXT, you can direct MASM to store the ASCII equivalent of a text string in memory. The assembler always packs two characters into each 16-bit word when it stores a text string (i.e., one character per 8-bit byte).

By default, the assembler packs character bytes left to right within memory words. You may change this convention by issuing the .TXTM pseudo-op.

If a string has an odd number of characters, the assembler places a null (all zero) byte in the word with the last character. If the string has an even number of characters, the assembler places either no null bytes or two null bytes after the last character, depending on your directions (see .TXTN).

If you wish to store only one or two ASCII characters in memory, you do not have to use pseudo-ops. "Special Integer-Generating Formats" in Chapter 2 describes alternative methods for storing characters.

Symbol Table

Symbol table pseudo-ops let you define machine instructions and user symbols. These pseudo-ops are shown in Table 6-10.

Table 6-10. Symbol Table Pseudo-Ops

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.DALC	YES	NO	Define an ALC instruction or expression
.DCMR	YES	NO	Define a commercial memory reference instruction or expression
.DEMR	YES	NO	Define an extended memory reference instruction or expression, without accumulator
.DERA	YES	NO	Define an extended memory reference instruction that requires an accumulator
.DEUR	YES	NO	Define an extended user instruction or expression
.DFLM	YES	NO	Define a floating-point load or store instruction or expression that requires an accumulator
.DFLS	YES	NO	Define a floating-point load or store status instruction that requires no accumulator
.DIAC	YES	NO	Define an I/O instruction requiring an accumulator
.DICD	YES	NO	Define an instruction requiring an accumulator and a count
.DIMM	YES	NO	Define an immediate-reference instruction requiring an accumulator
.DIO	YES	NO	Define an I/O instruction that does not use an accumulator
.DIOA	YES	NO	Define an I/O instruction that requires two fields

(continues)

Table 6-10. Symbol Table Pseudo-Ops

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.DISD	YES	NO	Define an instruction with source and destination accumulators and no skip
.DISS	YES	NO	Define an instruction with source and destination accumulators with skip
.DMR	YES	NO	Define a memory reference instruction with displacement and index
.DMRA	YES	NO	Define a memory reference instruction with two or three fields
.DUSR	YES	NO	Define a user symbol without implied formatting
.DXOP	YES	NO	Define an instruction with source, destination, and operation fields

(concluded)

All symbol table pseudo-ops (except .DUSR) name assembler instructions for 16-bit ECLIPSE computers. These instructions are described in the Programmer's Reference, ECLIPSE®-Line Computers manual. A system's parameter files use the symbol table pseudo-ops to define assembler instructions. The definitions reside in the MASM.PS (or MASM16.PS) disk file. This file is required to assemble source programs; it is described further in Chapter 8.

Symbol table pseudo-ops have the form:

$$\textbf{pseudo-op}\square\textbf{user-symbol} = \left\{ \begin{array}{l} \textbf{instruction} \\ \textbf{expression} \end{array} \right\}$$

where:

pseudo-op is a symbol table pseudo-op

user-symbol is a symbol assigned by the programmer

instruction and **expression** are as defined in Chapter 2

In symbol table pseudo-ops, a **user-symbol** is semipermanent. Its assembled value is the value of the instruction or expression following the equals sign.

Each symbol table pseudo-op (except .DUSR) defines a different type of instruction for 16-bit ECLIPSE computers. **User symbols** must be used with appropriate expressions. For example, the pseudo-op .DALC defines a symbol for an arithmetic and logical (ALC) instruction or expression. A symbol defined by the .DALC pseudo-op must be followed by expressions that represent the source and destination accumulators, and the ALC instruction's optional skip field. The format for a .DALC definition of a symbol, and the symbol as it would be used is:

$$\textbf{.DALC□user-symbol} = \left\{\begin{array}{l}\textbf{instruction}\\ \textbf{expression}\end{array}\right\}$$

•

•

user-symbol□expression□1□expression□2□<expression□3>

Where **expression 1, expression 2,** and the optional expression 3 are stored in the ALC instruction format as shown below.

0	1 2	3 4	5 6 7 8 9 10 11 12	13 14 15
	Expression 1	Expression 2		Expression 3

The symbol table pseudo-ops are described in Chapter 7. In summary, a symbol defined as semipermanent by symbol table pseudo-op must meet the following conditions.

- * As many expressions must follow the user-symbol as are required by the target format. Some formats permit optional expressions in addition to their required expressions. The Macroassembler generates a format (F) error if the number of expressions following a user-symbol do not meet the requirements of the target format.

- * Each expression must meet the width requirements of the target format's fields. For example, if

  expression□>□($2^{\text{field width}}$-1)

  the field is not changed and the Macroassembler generates an overflow (O) error.

- * If the field in which the expression is to be stored does not equal zero, the expression must equal zero. Otherwise, the field is not changed, and an overflow (O) error is generated.

A given user symbol that is defined in one symbol table pseudo-op can be redefined in another symbol table pseudo-op. The last definition will be the one assigned to the user symbol. A redefinition of a permanent symbol will result in a multiple definition (M) error if the /M function switch was used.

Symbol Deletion

The Macroassembler allows you to delete all symbol definitions, except pseudo-op definitions (the permanent symbols), from the current assembly.

Table 6-11 provides information about .XPNG, the pseudo-op you use to remove symbol definitions.

Table 6-11. Symbol Deletion Pseudo-Op

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.XPNG	YES	NO	Delete all semipermanent symbol definitions (instructions and macros) from the current assembly

You can use the .XPNG pseudo-op when you want to define your own symbol definitions in the permanent symbol table.

Miscellaneous

The pseudo-ops in this section do not fit into any of the other categories. Table 6-12 lists the miscellaneous pseudo-ops and provides a short description of each.

Table 6-12. Miscellaneous Pseudo-Ops

Pseudo-Ops	Assembler Directive	Value Symbol	Description
.FORC	YES	NO	Force Link to include this library module in the program file
.LMIT	YES	NO	Specifies partial binding of an object file
.OB	YES	NO	Name an object file
.PASS	NO	YES	Return a value corresponding to the current MASM pass number
.REV	YES	NO	Assign two revision level numbers to a program file
.TITL	YES	NO	Assign a name to a listing header
.TSK	YES	NO	Specify the number of tasks in your program

Refer to Chapter 7 for detailed descriptions of these pseudo-ops.

End of Chapter

Chapter 7
Pseudo-Op Descriptions

This chapter describes all the AOS MASM and AOS/VS MASM16 pseudo-ops. They are in alphabetic order for easy reference.

For each pseudo-op, we include

* the mnemonic that the Macroassembler recognizes (e.g., .NREL);

* the pseudo-op's title;

* the syntax of the pseudo-op as an assembler directive, if applicable;

* a functional description of the pseudo-op as an assembler directive (under "Purpose"), if applicable;

* a functional description of the pseudo-op as a value symbol (under "Value"), if applicable;

* one or more examples; and

* references for further information about related topics.

Coding Aids

Throughout this chapter, we use certain conventions and abbreviations to help you code each Macroassembler pseudo-op statement properly.

The Preface describes the notation we use to present pseudo-op statement syntax. Table 7-1 lists the abbreviations that we use in this chapter.

We explain all other abbreviations in the appropriate pseudo-op descriptions.

General References

Refer to the following sections of this manual for general information about pseudo-ops:

* An alphabetic listing of the pseudo-ops appears in Appendix B

- "Permanent Symbols" and "Pseudo-Ops" in Chapter 2 provide general information about pseudo-ops
- Chapter 6 presents and discusses each of the various categories of pseudo-ops
- "Symbol Interpretation" in Chapter 3 explains how the Macroassembler interprets the pseudo-op mnemonics that appear in your source module

Each pseudo-op description in this chapter also contains references specific to that pseudo-op.

Table 7-1. Abbreviations

Abbreviation	**Meaning**
□	Any combination of spaces, horizontal tabs, and/or one comma.
abs-expr	Absolute expression (see Chapter 3)
AC	Accumulator
ACD	Destination accumulator
ACS	Source accumulator
expr	Any macroassembler expression (see Chapters 2 and 3)
FPAC	Floating point accumulator
index	Addressing index (or mode)
instruction	ECLIPSE assembly language instruction
instr-symbol	Instruction symbol (see Chapter 2)
K	Approximately one thousand e.g., 1K words equals $1,024_{10}$ words
MRI	Memory reference instruction
MW	Megaword; i.e., 1 MW equals $1,048,576_{10}$ 16-bit words
user-symbol	User symbol (see Chapter 2)

(.)

Current location counter.

Value

The symbol . (period) has the value and relocation property of the current location counter. The location counter is an assembler variable that holds the address and relocation base of the next memory location the Macroassembler will assign.

Example

```
08 00000'020000        LDA 0,0         ;INSTRUCTION AT LOC 0
09 00001'040000        STA 0,0         ;INSTRUCTION AT LOC 1
10 00002'000002'       .               ;PSEUDO-OP (VALUE = LOCATION)
11       000010'       .LOC .+5        ;CHANGE LOCATION COUNTER
12                                     ;(CURRENT LOCATION PLUS FIVE)
13 00010'133000        ADD 1,2         ;INSTRUCTION AT LOCATION 10 (OCTAL)
```

References

"Location Counter" - Chapter 3
"Relocatability" - Chapter 3

.ARGCT

Number of arguments passed to macro.

Value

The pseudo-op .ARGCT is a value symbol. Its value equals the number of arguments you passed to the macro containing it. For example, if you pass three arguments to a macro, then the symbol .ARGCT has the value 3 for that macro expansion.

If you use .ARGCT outside a macro, its value is -1.

Examples

```
                    .MACRO A        ;MACRO THAT TAKES TWO ARGUMENTS
                    ^1+^2
                    .ARGCT          ;VALUE = NUMBER OF ARGUMENTS
                    %

                    A 4,5
00000'000011        4+5
00001'000002        .ARGCT          ;VALUE = NUMBER OF ARGUMENTS
```

```
                    .MACRO ARG      ;DEFINE MACRO 'ARG'
                    .IFE  .ARGCT    ;IF YOU CALL 'ARG' WITH NO
                    10              ;ARGUMENTS, ASSEMBLE THE
                    .ENDC           ;VALUE 10.  OTHERWISE,
                    .IFN  .ARGCT    ;ASSEMBLE THE VALUE OF THE
                    ^1              ;FIRST ARGUMENT.
                    .ENDC
                    %

                    ARG             ;CALL 'ARG' WITHOUT ARGUMENTS
      000001        .IFE  .ARGCT    ;IF YOU CALL 'ARG' WITH NO
00000'000010        10              ;ARGUMENTS, ASSEMBLE THE
                    .ENDC           ;VALUE 10.  OTHERWISE,
      000000        .IFN  .ARGCT    ;ASSEMBLE THE VALUE OF THE
                                    ;FIRST ARGUMENT.
                    .ENDC

                    ARG 2           ;CALL 'ARG' WITH ONE ARGUMENT
      000000        .IFE  .ARGCT    ;IF YOU CALL 'ARG' WITH NO
                    10              ;ARGUMENTS, ASSEMBLE THE
                    .ENDC           ;VALUE 10.  OTHERWISE,
      000001        .IFN  .ARGCT    ;ASSEMBLE THE VALUE OF THE
00001'000002        2               ;FIRST ARGUMENT.
                    .ENDC
```

References

"Macros" - Chapter 5
"Macro-Related Pseudo-Ops" - Chapter 5

.ASYM

Define an accumulating symbol.

Syntax

.ASYM□user-symbol

Purpose

This pseudo-op defines a **user-symbol** whose value will be the sum of the values assigned to it by all modules in which it is declared. For this summing to take place, **user-symbol** must be declared by an .ASYM and assigned a value. To access its accumulated value it must be identified to Link by an .EXTN pseudo-op in that module. You cannot use both .ASYM and .EXTN with the same user symbol in the same module. Its intermediate values are not accessible. The relocation type of this symbol is preserved.

Example

```
                    .ASYM A      Module 1
                    A=1
```

```
                    .ASYM A      Module 2
                    A=2
```

```
                    .EXTN A      Module 3
                    A
```

* This value is resolved only after the modules have been linked.

Reference

"Intermodule Communication" - Chapter 6

.BLK

Reserve a block of memory.

Syntax

.BLK□abs-expr

Purpose

The .BLK pseudo-op reserves a block of memory words. **Abs-expr** specifies the length (in 16-bit words) of this block. **Abs-expr** must be a positive absolute expression.

The assembler increments the current location counter by **abs-expr** when it encounters .BLK in your source.

Example

```
00000'122250         FSTS 0,ACLOC         ;STORE THE FLOATING POINT
      000010
00002'126250         FSTS 1,ACLOC+2       ;ACCUMULATORS IN CONSECUTIVE
      000010
00004'132250         FSTS 2,ACLOC+4       ;LOCATIONS STARTING AT
      000010
00006'136250         FSTS 3,ACLOC+6       ;ADDRESS ACLOC.
      000010
00010'000411          JMP     SKIPP       ;JUMP AROUND DATA BLOCK
00011'000010 ACLOC:   .BLK 10             ;SAVE 10 (OCTAL) WORDS OF
                                          ;MEMORY.
00021'020150 SKIPP:    LDA 0,150          ;NOTE THAT THE LOCATION COUNTER
                                          ;JUMPS 10 WORDS.
```

References

"Absolute Expressions" - Chapter 3
"Assigning Locations" - Chapter 3

.COMM

Reserve a labeled common area.

Syntax

.COMM□user-symbol□abs-expr

Purpose

The .COMM pseudo-op reserves a labeled (or named) common area for intermodule communication. A common area is a data storage area that you may access from separately assembled modules in your program.

The Macroassembler assigns the name **user-symbol** to this common area. MASM regards **user-symbol** as an entry point and, therefore, you should not redefine this symbol anywhere in your program.

Specify the size of the common area (in 16-bit words) in the **abs-expr** argument. This argument must be a positive absolute expression.

To reference this common area from another module in your program, use .COMM, .EXTN, or .EXTU to declare **user-symbol** as externally defined. If you issue the same .COMM statement in two separately assembled modules, Link resolves them to the same area in memory.

Example

```
.TITL A         ;MODULE 'A'
.NREL
.COMM X,100     ;RESERVE A 100 WORD COMMON AREA
                ;NAMED 'X'
.COMM Y,50      ;RESERVE A 50 WORD COMMON AREA
                ;NAMED 'Y'
.END
.TITL B         ;SEPARATELY ASSEMBLED MODULE 'B'
.NREL
.COMM X,100     ;'X' REFERS TO THE SAME COMMON AREA
                ;DECLARED IN MODULE 'A.'
.END
.TITLE C        ;SEPARATELY ASSEMBLED MODULE 'C'
.NREL
.EXTN X         ;'X' REFERS TO THE STARTING ADDRESS
                ;OF THE COMMON AREA DECLARED IN 'A.'
.END
```

References

"Absolute Expressions" - Chapter 3
AOS Link User's Manual or
AOS/VS Link and Library File Editor User's Manual
"Intermodule Communication" - Chapter 6
"User Symbols" - Chapter 2

.CSIZ

Reserve an unlabeled common area.

Syntax

.CSIZ□abs-expr

Purpose

The .CSIZ pseudo-op reserves an unlabeled common area for intermodule communication. A common area is a data storage area that you may access from separately assembled modules in your program.

The **abs-expr** argument is the size in 16-bit words of the unlabeled common area. This argument must be a positive absolute expression.

If you include more than one .CSIZ pseudo-op in a single source module, MASM uses the largest value as the size of the unlabeled common area. Similarly, if separately assembled modules issue .CSIZ pseudo-ops, Link uses the largest value.

Example

```
.TITLE A
.CSIZ  20          ;20 WORDS ALLOCATED
   .               ;TO UNLABELED COMMON
   .
   .
.END
.TITLE  X
.CSIZ  50          ;50 WORDS ALLOCATED
  .                ;TO UNLABELED COMMON
  .                ;TO BE SHARED WITH PROGRAM A.
  .
.END
```

References

"Absolute Expressions" - Chapter 3
AOS Link User's Manual or
AOS/VS Link and Library File Editor User's Manual
"Intermodule Communication" - Chapter 6

.DALC

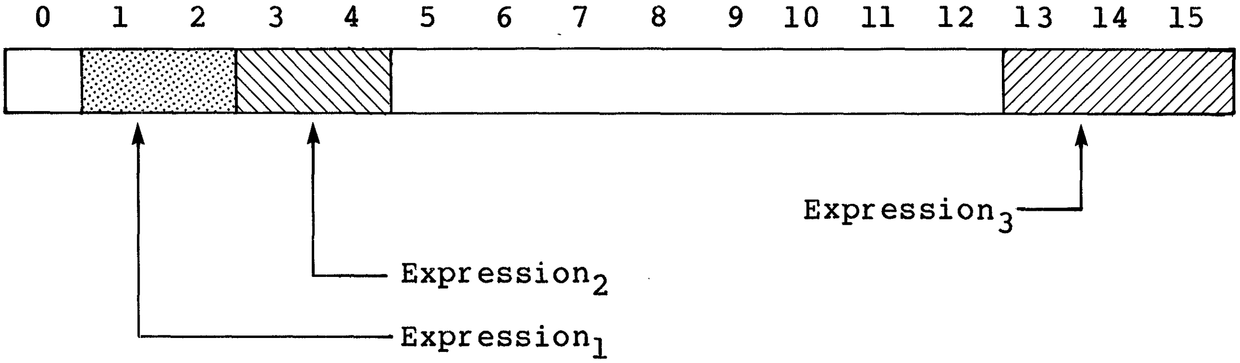
Define ALC instruction.

Syntax

```
                    ( instruction )
.DALC□user-symbol= {               }
                    ( expression  )
```

Purpose

The .DALC pseudo-op defines **user-symbol** as a semipermanent symbol having the value of **instruction** or **expression.** This **user-symbol** implies the format of an ALC instruction. At least two fields, and optionally, three, are required. These fields are assembled as shown below.



The atom # (no-load indicator) may be used anywhere as a break character. It assembles a 1 at bit position 12.

A given **user-symbol** defined in one .DALC pseudo-op may be redefined in another .DALC pseudo-op. The last definition will be the one assigned to **user-symbol.**

A three-character symbol defined by this pseudo-op can have carry and shift flags. These flags are concatenated to the right side of the symbol in the following format:

user-symbol<carry><shift>

The carry flag must be L, R, or S and the shift flag must be Z, O, or C. The flags are optional. They set bits 8-9, 10-11 as follows:

.DALC (continued)

Mnemonic	Shift Bits (8-9)	Carry Bits (10-11)	Description
L	01	---	Left shift
R	10	---	Right shift
S	11	---	Swap
Z	---	01	Set carry bit to 0
O	---	10	Set carry bit to 1
C	---	11	Complement carry bit

Example

```
08 00000'103000   ADD 0,0            ;STANDARD ECLIPSE
09                                   ;'ADD' INSTRUCTION
10
11 00001'157001   ADD 2,3,SKP        ;'ADD' INSTRUCTION WITH
12                                   ;SKIP FIELD
13
14       103000   .DALC DDA=103000   ;'DDA' DEFINED TO BE
15                                   ;SAME AS 'ADD'
16
17 00002'103000   DDA 0,0            ;'DDA' INSTRUCTION (NOTE ASSEMBLED WORD)
18 00003'157001   DDA 2,3,SKP        ;'DDA' WITH SKIP FIELD
19
20 00004'133120   ADDZL 1,2          ;'ADD' WITH SHIFT AND CARRY BITS SET
21 00005'133120   DDAZL 1,2          ;'DDA' WITH SHIFT AND CARRY BITS SET
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DCMR

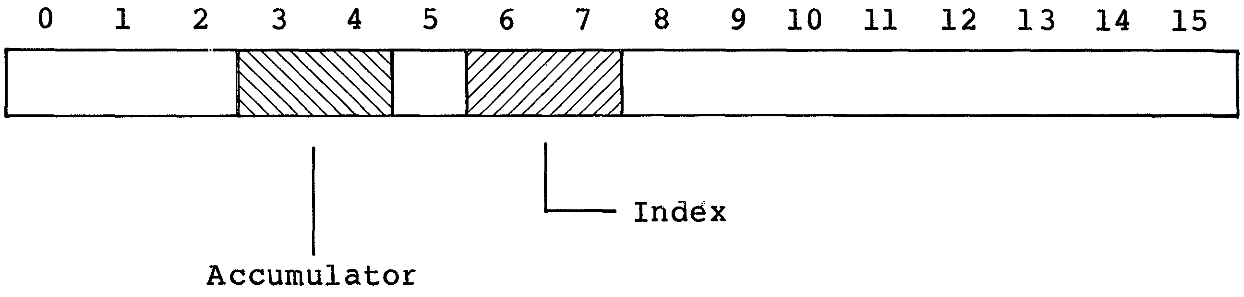
Define commercial memory reference instruction.

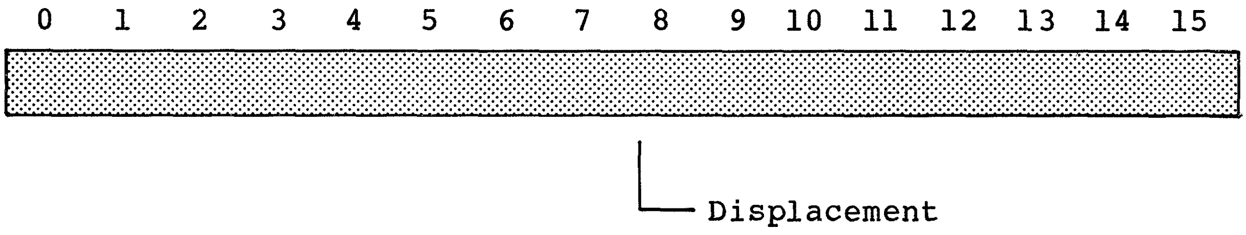
Syntax

$$\textbf{.DCMR}\square\textbf{user-symbol} = \left\{ \begin{array}{l} \textbf{instruction} \\ \textbf{expression} \end{array} \right\}$$

Purpose

This pseudo-op defines **user-symbol** as a semipermanent extended commercial memory reference symbol having the value of **instruction** or **expression.** This symbol implies the format of an instruction that requires an accumulator and displacement. It also permits an optional index. The fields are assembled as shown below.





The format for using this symbol is:

semipermanent-symbol□accumulator□displacement<□<u>index</u>>

.DCMR (continued)

Example

```
06 00000 122170   ESTB 0,0       ;EXTENDED STORE BYTE INSTRUCTION
07       000000
08
09       122170   .DCMR BTSE=122170  ;DEFINE 'BTSE' SAME AS 'ESTB'
10
11 00002 122170   BTSE 0,0           ;'BTSE' INSTRUCTION
12       000000
13
14
15 00004 126170   ESTB 1,CON1   ;'ESTB' INSTRUCTION WITH AC1 AND DISPLACEMENT
16       000666
17 00006 126170   BTSE 1,CON1   ;'BTSE' INSTRUCTION WITH AC1 AND DISPLACEMENT
18       000666
19
20
21       000666   CON1=  666
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DEMR

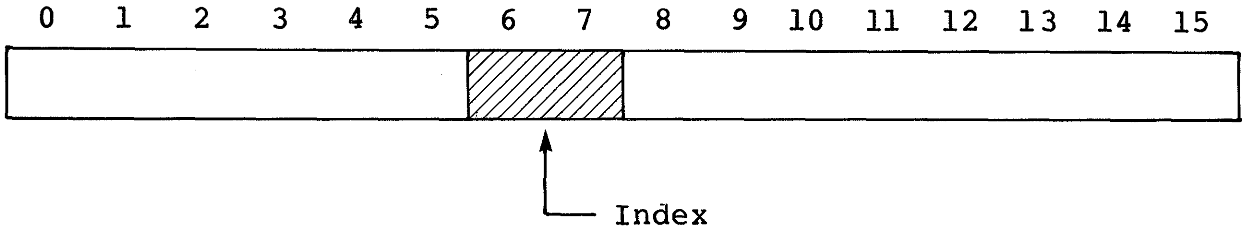
Define extended memory reference instruction.

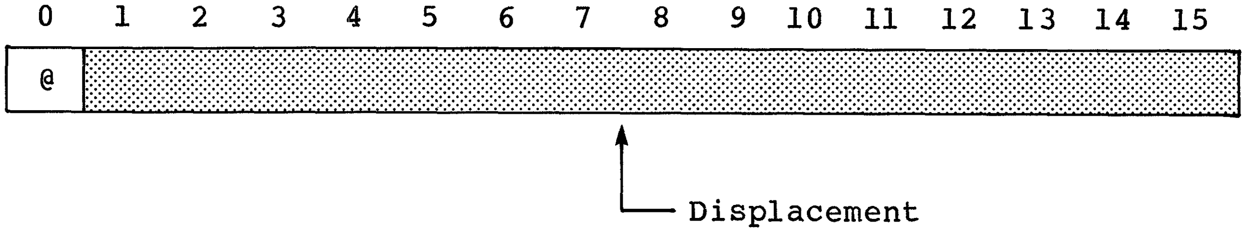
Syntax

.DEMR□user-symbol = $\left\{\begin{array}{l}\textbf{instruction}\\ \textbf{expression}\end{array}\right\}$

Purpose

This pseudo-op defines **user-symbol** as a semipermanent extended memory reference symbol and gives it the value of **instruction** or **expression.** This symbol implies the format of an instruction that does not require an accumulator. One field is required; an index is optional. They are assembled as shown below.





The formats for using this semipermanent symbol are:

semipermanent-symbol□displacement

semipermanent-symbol□displacement□index

The displacement and index fields are set according to the format used and the set of addressing rules described in Chapter 3.

The @ atom (indirection flag) may be coded on the source line as a break character. If present, a 1 is assembled in bit 0 of the second word.

.DEMR (continued)

Example

```
08 00000'106470          EJSR LBL1        ;EXTENDED JUMP TO SUBROUTINE
09       000013
10
11
12
13       106070          .DEMR RSJE=106070 ;DEFINE 'RSJE' SAME AS 'EJSR'
14
15 00002'106470          RSJE LBL1        ;NEWLY DEFINED INSTRUCTION
16       000011
17
18 00004'106470          EJSR @LBL1       ;'EJSR' WITH INDIRECT BIT SET
19       100007
20 00006'106470          RSJE @LBL1       ;'RSJE' WITH INDIRECT BIT SET
21       100005
22
23 00010'107470          EJSR LBL1,3      ;'EJSR' INDEXED WITH AC3
24       000014'
25 00012'107470          RSJE LBL1,3      ;'RSJE' INDEXED WITH AC3
26       000014'
27
28             LBL1:                      ;ADDRESS OF SUBROUTINE
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DERA

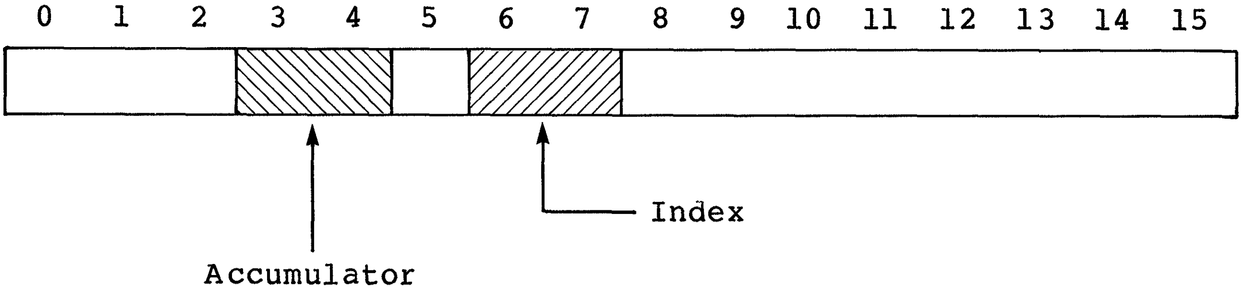
Define extended memory reference instruction.

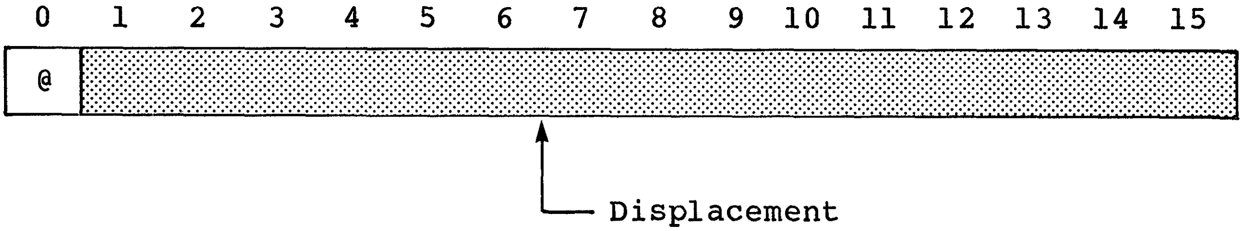
Syntax

$$\textbf{.DERA}\square\textbf{user-symbol} = \left\{ \begin{array}{l} \textbf{instruction} \\ \textbf{expression} \end{array} \right\}$$

Purpose

This pseudo-op defines **user-symbol** as a semipermanent extended memory reference symbol having the value of **instruction** or **expression.** This symbol implies the format of an instruction that requires an accumulator. Two fields are required and index is optional. They are assembled as shown.





The formats for using this symbol are:

semipermanent-symbol□accumulator□displacement

semipermanent-symbol□accumulator□displacement□index

The displacement and index fields are set according to the format used and the set of addressing rules described in Chapter 3.

.DERA (continued)

The @ atom (indirection flag) may be coded on the source line as a break character. If present, a 1 is assembled in bit 0 of the second word.

Example

```
08 00000'142070    ESTA 0,0            ;EXTENDED STORE INSTRUCTION
09       000000
10
11       142070    .DERA ASTE=142070   ;'ASTE' DEFINED AS 'ESTA'
12
13 00002'142070    ASTE 0,0            ;NEWLY DEFINED INSTRUCTION
14       000000
15
16
17 00004'146070    ESTA 1,3            ;'ESTA' AC=1, DISPLACEMENT=3
18       000003
19 00006'146070    ASTE 1,3            ;'ASTE' AC=1, DISPLACEMENT=3
20       000003
21
22 00010'152470    ESTA 2,@2,1         ;'ESTA' INDIRECT WITH INDEX
23       100002
24 00012'152470    ASTE 2,@2,1         ;'ASTE' INDIRECT WITH INDEX
25       100002
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DEUR

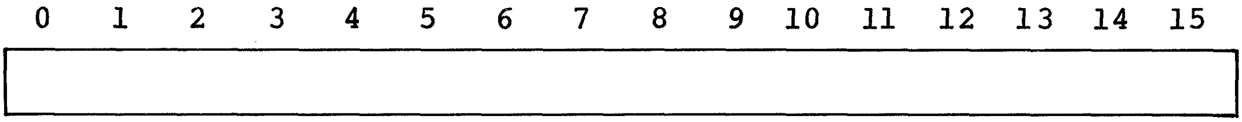
Define extended user instruction.

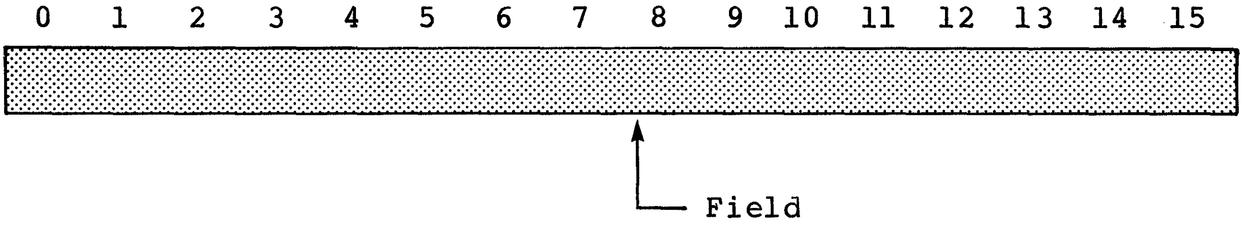
Syntax

.DEUR□user-symbol = $\left\{\begin{array}{l}\textbf{instruction}\\ \textbf{expression}\end{array}\right\}$

Purpose

This pseudo-op defines **user-symbol** as a semipermanent extended symbol having the value of **instruction** or **expression.** The expression can be either an expression, an external normal, or an external displacement. This symbol implies the format of an instruction that does not require an accumulator. One field is required and is assembled as shown.





The format for this symbol is:

semipermanent-symbol□expression

Example

```
08 00000'163710      SAVE 0              ;'SAVE' INSTRUCTION
09       000000
10
11       163710      .DEUR EVAS=163710   ;'EVAS' DEFINED AS 'SAVE'
12
13 00002'163710      SAVE 3              ;'SAVE' WITH FRAME-SIZE 3
14       000003
15 00004'163710      EVAS 3              ;'EVAS' WITH FRAME-SIZE 3
16       000003
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DFLM

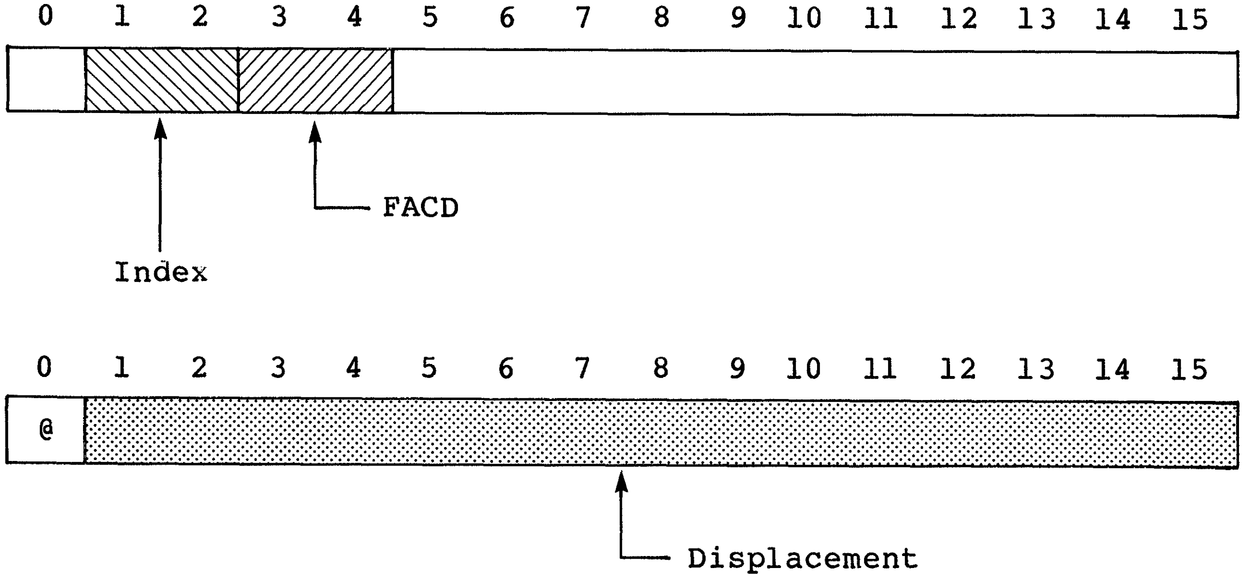
Define floating load or store instruction.

Syntax

.DFLM□user-symbol = { **instruction** | **expression** }

Purpose

This pseudo-op defines **user-symbol** as a semipermanent floating load or store memory reference symbol having the value of **instruction** or **expression.** This symbol implies the format of an instruction that requires an accumulator. Two fields are required and one is optional. They are assembled as shown.



The format for this semipermanent symbol is:

semipermanent-symbol□accumulator□displacement<□index>

.DFLM (continued)

Example

```
00000'101150     FAMD 0,0        ;FLOATING DOUBLE-PRECISION ADD TO MEMORY
      000000

      101150     .DFLM DMAF=101150   ;DEFINE 'DMAF' AS 'FAMD'

00002'101150     DMAF 0,0
      000000

00004'131150     FAMD 2,@5,1         ;'FAMD' WITH INDIRECTION AND
      100005
                                     ;PC-RELATIVE ADDRESSING

00006'131150     DMAF 2,@5,1         ;'DMAF' WITH INDIRECTION AND
      100005
                                     ;PC-RELATIVE ADDRESSING
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DFLS

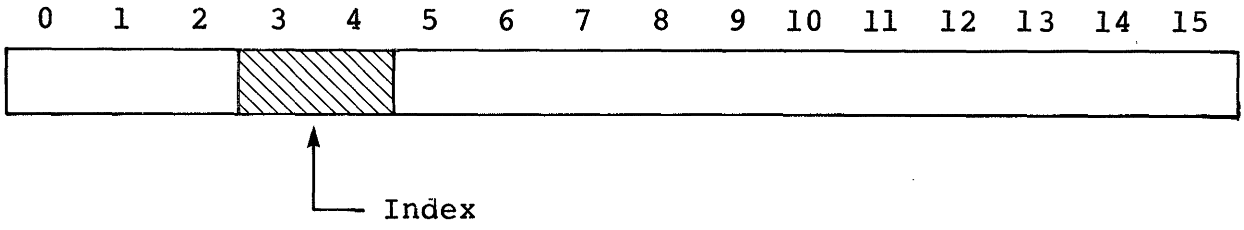
Define floating point status instruction.

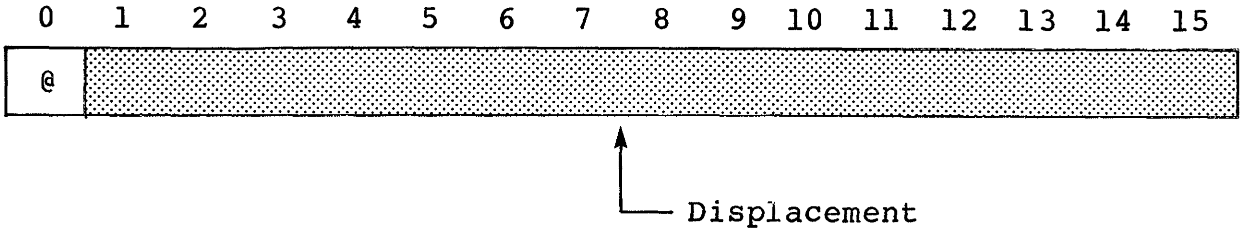
Syntax

.DFLS□user-symbol = { **instruction** / **expression** }

Purpose

This pseudo-op defines **user-symbol** as a semipermanent floating-point status register reference symbol having the value of **instruction** or **expression.** This symbol implies the format of an instruction that does not require an accumulator. One field is required; an index is optional. These fields are assembled as shown:





The format for using this symbol is:

semipermanent-symbol□displacement<□index>

.DFLS (continued)

Example

```
08 00000'103350   FSST 0              ;STORE FLOATING-POINT STATUS
09        000000
10
11        103350   .DFLS TSSF=103350  ;DEFINE 'TSSF' AS 'FSST'
12
13 00002'103350   TSSF 0              ;NEWLY DEFINED INSTRUCTION
14        000000
15
16 00004'113350   FSST @100,2         ;'FSST' WITH INDIRECTION AND INDEX
17        100100
18 00006'113350   TSSF @100,2         ;'TSSF' WITH INDIRECTION AND INDEX
19        100100
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DIAC

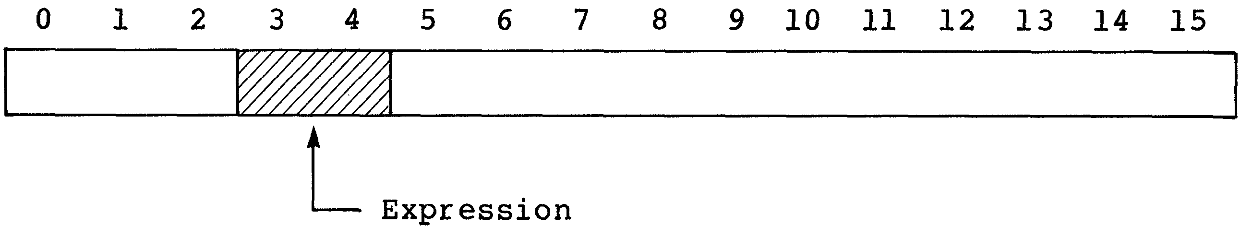
Define an instruction requiring an accumulator.

Syntax

$$\textbf{.DIAC}\square\textbf{user-symbol} = \left\{ \begin{array}{l} \textbf{instruction} \\ \textbf{expression} \end{array} \right\}$$

Purpose

This pseudo-op defines **user-symbol** and gives it the value of **instruction** or **expression.** This symbol implies the format of an instruction requiring an accumulator. One field is required. It is assembled as shown below.



The format for using this symbol is:

user-symbol□expression

Example

```
08 00000'123370         XCT 0           ;EXECUTE INSTRUCTION
09
10       123370         .DIAC TCX=123370 ;DEFINE TCX AS XCT
11
12 00001'137370         XCT 3           ;XCT WITH ACCUMULATOR 3
13 00002'137370         TCX 3           ;TCX WITH ACCUMULATOR 3
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DICD

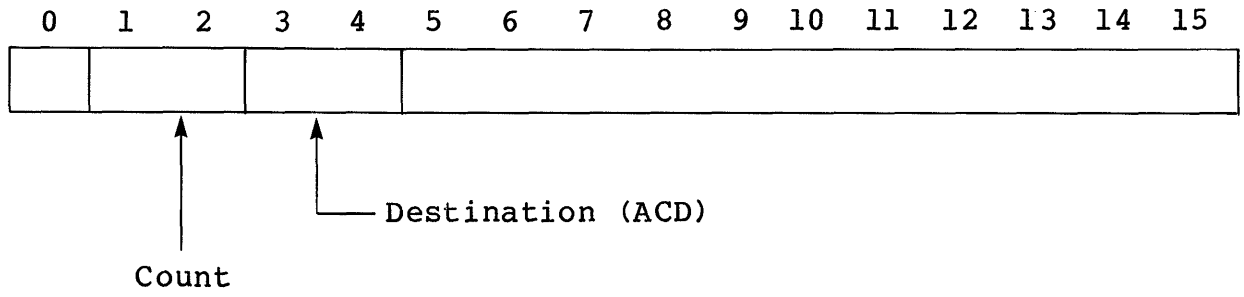
Define an instruction requiring an accumulator and count.

Syntax

$$\textbf{.DICD□user-symbol} = \left\{ \begin{array}{l} \textbf{instruction} \\ \textbf{expression} \end{array} \right\}$$

Purpose

This pseudo-op defines **user-symbol** as a semipermanent symbol having count and destination fields. The symbol has the value of **instruction** or **expression.** This symbol implies the format of an instruction that requires an accumulator and a count from 1 to 4. Two fields are required and are assembled as shown.



The format for using this symbol is:

semipermanent-symbol□count□destination-ac

The count is computed as

count = specified_value - 1

Thus, the range of permitted values is 1 - 4.

Example

```
08 00000'100110     SBI 1,0          ;SUBTRACT IMMEDIATE INSTRUCTION
09
10       100110     .DICD IBS=100110  ;DEFINE 'IBS' AS 'SBI'
11
12 00001'100110     IBS 1,0          ;NEWLY DEFINED INSTRUCTION
13
14 00002'124110     SBI 2,1          ;'SBI' WITH COUNT AND ACCUMULATOR 1
15 00003'124110     IBS 2,1          ;'IBS' WITH COUNT AND ACCUMULATOR 1
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DIMM

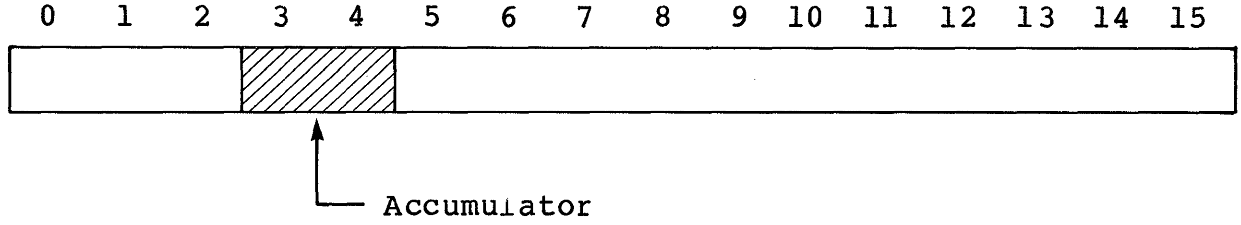
Define an instruction requiring an accumulator and an immediate word.

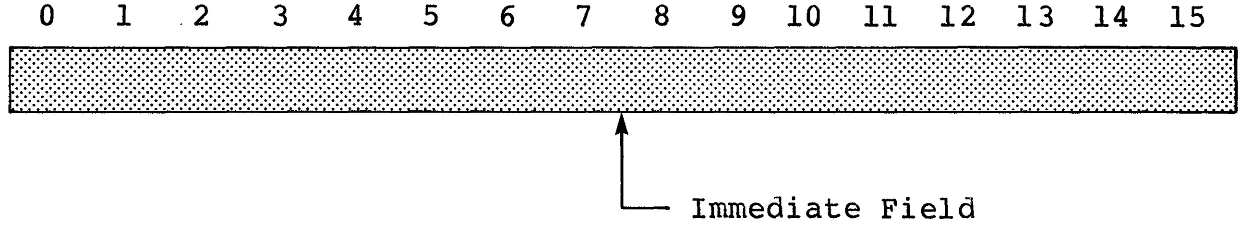
Syntax

$\textbf{.DIMM□user-symbol} = \left\{ \begin{array}{l} \textbf{instruction} \\ \textbf{expression} \end{array} \right\}$

Purpose

This pseudo-op defines **user-symbol** as a semipermanent, immediate-reference symbol having the value of **instruction** or **expression.** This symbol implies the format of an instruction that requires an accumulator and a 16-bit immediate word. Two fields are required and are assembled as shown.





The format for using this symbol is:

semipermanent-symbol□immediate-value□destination-ac

Example

```
08 00000'163770     ADDI 0,0        ;EXTENDED ADD IMMEDIATE INSTRUCTION
09       000000
10
11       163770     .DIMM IDDA=163770 ;DEFINE 'IDDA' AS 'ADDI'
12
13 00002'163770     IDDA 0,0        ;NEWLY DEFINED INSTRUCTION
14       000000
15
16 00004'173770     ADDI 377,2      ;'ADDI' WITH IMMEDIATE FIELD VALUE
17       000377
18 00006'173770     IDDA 377,2      ;'IDDA' WITH IMMEDIATE FIELD VALUE
19       000377
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DIO

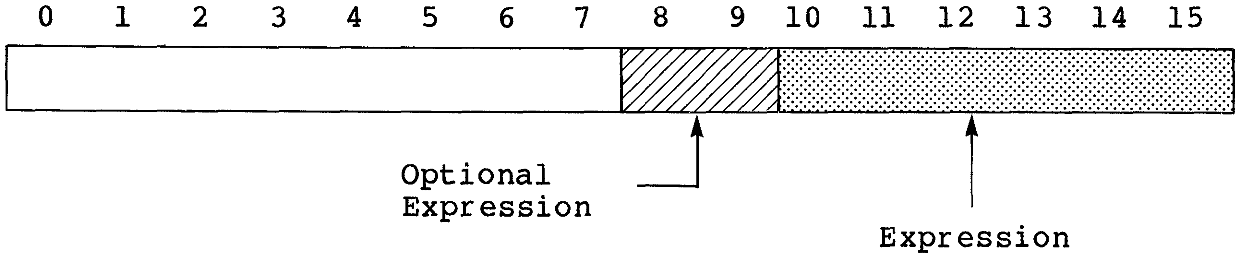
Define an I/O instruction without an accumulator.

Syntax

.DIO□user-symbol<optional expression> = $\left\{\begin{array}{l}\textbf{instruction}\\\textbf{expression}\end{array}\right\}$

Purpose

This pseudo-op defines **user-symbol** as a semipermanent symbol having the value of **instruction** or **expression.** This symbol implies the format of an I/O instruction without an AC field. One field is required; it is assembled as shown below.



The format for using this symbol is:

user-symbol<optional expression> **expression**

A three-character symbol defined by this pseudo-op can have a flag (optional expression in the above format) that sets the Busy/Done bits for a device. This flag is concatenated to the right of the user symbol. Each letter represents a Busy/Done combination and sets bits 8-9 of the instruction word as follows:

Mnemonic	Bits 8-9	Description
S	01	Starts device (clears Done, sets Busy)
C	10	Idles device (clears Done and Busy)
P	11	Pulses bus control line (sets Done and Busy)

Example

```
08 00000'063400         SKPBN 0          ;'I/O SKIP' INSTRUCTION
09
10       063400         .DIO PKS=063400  ;'PKS' DEFINED AS 'SKP'
11
12 00001'063400         PKS 0            ;NEWLY DEFINED INSTRUCTION
13
14 00002'063510         SKPBZ,10         ;SKIP ON TTY-IN BUSY=0
15 00003'063510         PKSS,10          ;SKIP ON TTY-IN BUSY=0
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DIOA

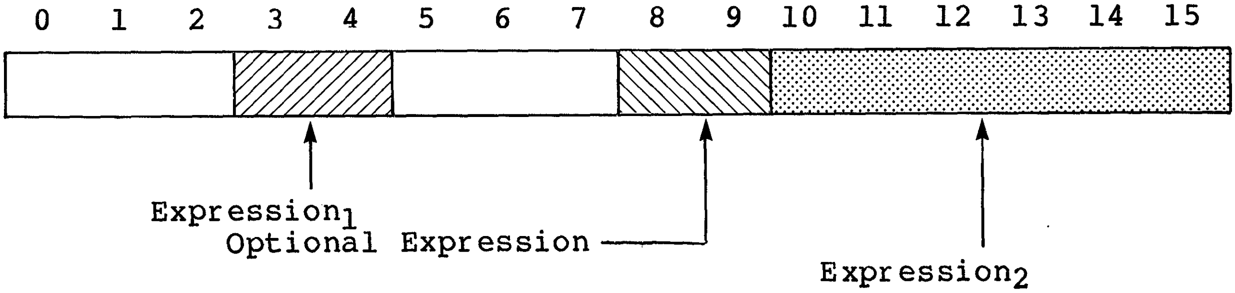
Define an I/O instruction with two required fields.

Syntax

.DIOA□**user-symbol**<optional expression> = $\left\{\begin{matrix}\textbf{instruction}\\\textbf{expression}\end{matrix}\right\}$

Purpose

This pseudo-op defines **user-symbol** as a semipermanent symbol having the value of **instruction** or **expression.** This symbol implies the format of an I/O instruction with two required fields. The fields are assembled as shown below.



The format for using this symbol is:

user-symbol<optional expression>□**expression**$_1$
□**expression**$_2$

A three-character symbol defined by this pseudo-op can have a flag (optional expression in the above format) that sets the Busy/Done bits for a device. This flag is concatenated to the right of the user symbol. Each letter represents a Busy/Done combination and sets bits 8-9 of the instruction word as follows:

Mnemonic	**Bits 8-9**	**Description**
S	01	Starts device (clears Done, sets Busy)
C	10	Idles device (clears Done and Busy)
P	11	Pulses bus control line (sets Done and Busy)

Example

```
08 00000'062400         DIC 0,0         ;DATA IN C INSTRUCTION
09
10       062400         .DIOA CID=062400 ;'CID' DEFINED AS 'DIC'
11
12 00001'076610         DICC 3,10       ;READ TTY REG C INTO AC3
13                                      ;AND SET INTERRUPT-ON = 0
14
15 00002'076610         CIDC 3,10       ;READ TTY REG C INTO AC3
16                                      ;AND SET INTERRUPT-ON = 0
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DISD

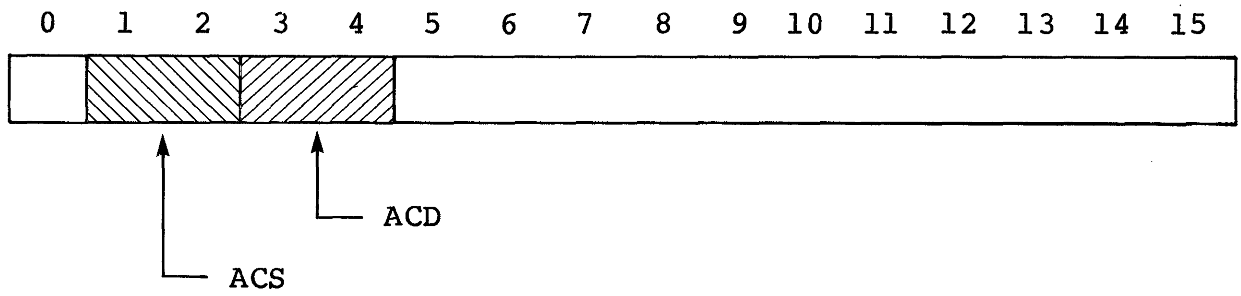
Define an instruction with source and destination accumulators.

Syntax

$$\textbf{.DISD}\square\textbf{user-symbol} = \left\{ \begin{array}{l} \textbf{instruction} \\ \textbf{expression} \end{array} \right\}$$

Purpose

This pseudo-op defines **user-symbol** as a semipermanent reference symbol with source and destination fields; it does not allow the no-load flag or skip conditions. The instruction cannot cause a skip. The symbol has the value of **instruction** or **expression.** This symbol implies the format of an instruction that requires a source and a destination accumulator. Two fields are required and are assembled as shown.



The format for using this symbol is:

semipermanent-symbol□source-ac□destination-ac

Example

```
08 00000'100250        FSS 0,0          ;SINGLE PRECISION SUBTRACT
09
10       100250        .DISD SSF=100250 ;'SSF' DEFINED AS 'FSS'
11
12 00001'100250        SSF 0,0          ;NEWLY DEFINED INSTRUCTION
13
14 00002'154250        FSS 2,3          ;SUBTRACT AC2 FROM AC3
15 00003'154250        SSF 2,3          ;SUBTRACT AC2 FROM AC3
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DISS

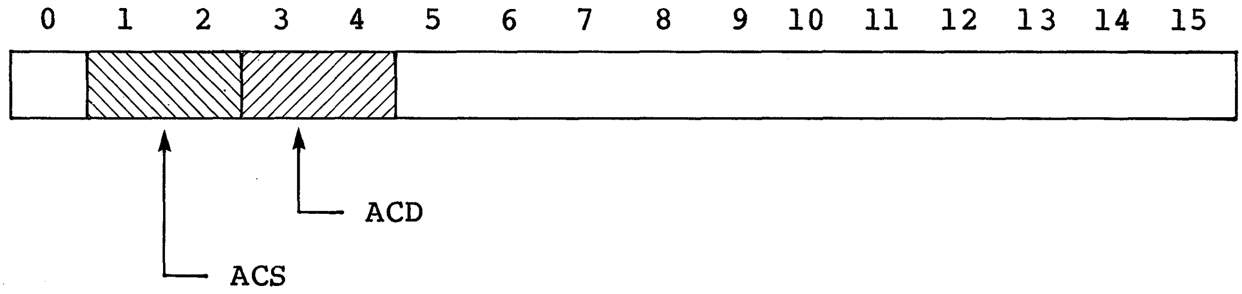
Define an instruction with source and destination accumulator allowing skip.

Syntax

$$\text{.DISS□user-symbol} = \left\{ \begin{array}{c} \textbf{instruction} \\ \textbf{expression} \end{array} \right\}$$

Purpose

This pseudo-op defines **user-symbol** as a semipermanent reference symbol with source and destination fields. The no-load flag cannot be used and no skip condition can be specified. However, the instruction may cause a skip to occur. The .DISS symbols differ from the .DISD symbols in that .DISS symbols may cause a skip and .DISD symbols never cause a skip. The symbol has the value **instruction** or **expression.** This symbol implies the format of an instruction that requires a source and destination accumulator. Two fields are required and are assembled as shown.



The format for using this symbol is:

semipermanent-symbol□source-ac□destination-ac

Example

```
08 00000'101110      SGE 0,0           ;SKIP IF SOURCE-AC GREATER
09                                     ;THAN DESTINATION-AC
10
11       101110      .DISS EGS=101110 ;'EGS' DEFINED AS 'SGE'
12
13 00001'101110      EGS 0,0           ;NEWLY DEFINED INSTRUCTION
14
15 00002'171110      SGE 3,2           ;SKIP IF CONTENTS OF AC3 GREATER
16                                     ;THAN CONTENTS OF AC2
17 00003'171110      EGS 3,2           ;SKIP IF CONTENTS OF AC3 GREATER
18                                     ;THAN CONTENTS OF AC2
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DMR

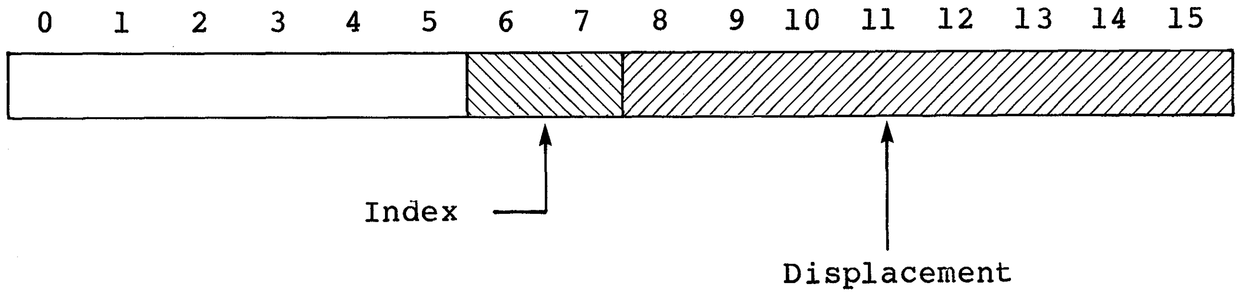
Define a memory reference instruction with displacement and index.

Syntax

.DMR□user-symbol = $\left\{ \begin{array}{l} \textbf{expression} \\ \textbf{instruction} \end{array} \right\}$

Purpose

This pseudo-op defines **user-symbol** as a semipermanent symbol having the value of **instruction** or **expression.** This symbol implies the format of an MR instruction with either one or two required fields (an address or a displacement and index). The fields are assembled as shown below.



The formats for using this symbol are:

semipermanent-symbol□displacement

semipermanent-symbol□displacement□index

The displacement and index fields are set according to the format used and the MRI addressing rules described in Chapter 3.

The atom @ (indirection flag) may be coded anywhere in the source line as a break character. This atom assembles a 1 at bit position 5.

Example

```
08 00000'004000        JSR 0,0            ;JUMP TO A SUBROUTINE
09
10       004000        .DMR RSJ=004000 ;RSJ DEFINED AS JSR
11
12 00001'004000        RSJ 0,0            ;NEWLY DEFINED INSTRUCTION
13
14
15 00002'007077        JSR @77,2          ;JUMP TO A SUBROUTINE WITH
16                                        ;INDIRECTION AND INDEX
17
18 00003'007077        RSJ @77,2          ;JUMP TO A SUBROUTINE WITH
19                                        ;INDIRECTION AND INDEX
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DMRA

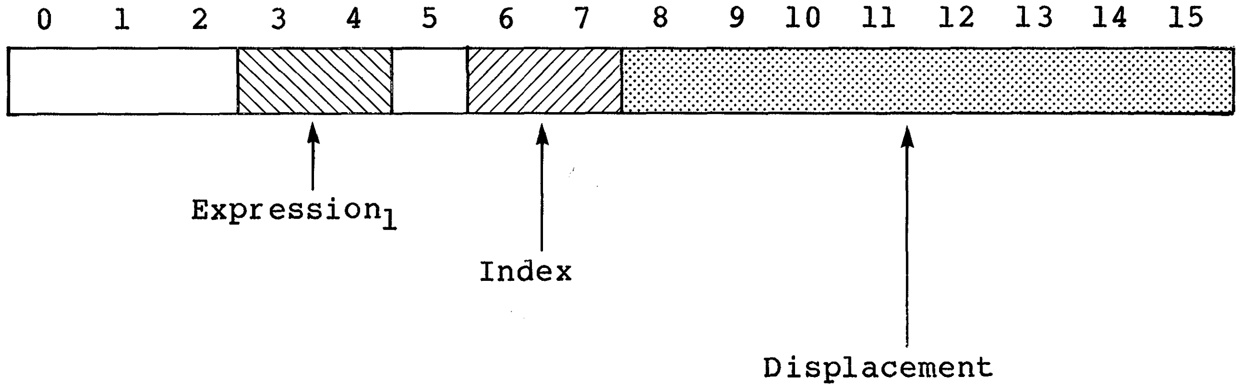
Define a memory reference instruction with two or three fields.

Syntax

$$\textbf{.DMRA□user-symbol} = \left\{\begin{matrix}\textbf{expression}\\ \textbf{instruction}\end{matrix}\right\}$$

Purpose

This pseudo-op defines **user-symbol** as a semipermanent symbol having the value of **instruction** or **expression.** This symbol implies the format of an memory reference instruction with either two or three required fields. The first field specifies an accumulator. Where there are two fields, the second is an implied address. Where there are three fields, the second and third fields are displacement and index respectively. The fields are assembled as shown below.



The formats for using this symbol are:

user-symbol□expression$_1$**□expression**$_2$

user-symbol□expression$_1$**□expression**$_2$
□expression$_3$

The displacement and index fields are set according to the format chosen and MRI addressing rules described in Chapter 3.

The atom @ (indirection flag) may be coded anywhere in the source line as a break character. This atom assembles a 1 at bit position 5.

Example

```
08 00000'020000     LDA 0,0           ;LOAD ACCUMULATOR
09
10       020000     .DMRA ADL=020000 ;DEFINE ADL AS LDA
11
12 00001'020000     ADL 0,0           ;NEWLY DEFINED INSTRUCTION
13
14
15 00002'033433     LDA 2,@33,3       ;LOAD AC2 USING AC3 AS THE INDEX
16                                    ;AND INDIRECTION
17
18 00003'033433     ADL 2,@33,3       ;LOAD AC2 USING AC3 AS THE INDEX
19                                    ;AND INDIRECTION
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.DO

Assemble source lines repetitively.

Syntax

.DO□abs-expr

Purpose

The .DO pseudo-op directs MASM to assemble a portion of your source module repetitively. MASM assembles the source lines following .DO the number of times given in **abs-expr. Abs-expr** must be an absolute expression.

You must terminate each .DO loop with the .ENDC pseudo-op. .ENDC can take an argument (see the .ENDC description). Thus, the .DO portion of your module has the general form:

```
.DO□abs-expr
       .                 ;MASM assembles these
       .                 ;lines abs-expr times.
       .
.ENDC                    ;Terminates the .DO loop.
```

You may use .DO to perform conditional assembly of source lines by passing a relational expression as an argument (pass an expression that contains <, >, ==, <=, >=, or <>). If the relational expression is true, its value is 1 and MASM assembles the .DO loop once. If the relational expression is false, its value is 0 and MASM does not assemble the loop.

You may nest .DOs to any depth. Be sure the innermost .DO corresponds with the innermost .ENDC, etc.

You must place the .ENDC pseudo-op at the same source level as the .DO pseudo-op or MASM will report an error and ignore the .DO statement. See "Loops and Conditionals in Macros" (Chapter 5) for more information.

NOTE: There is an implementation restriction that the .ENDC that terminates a .DO must be encountered before the end of the source file that contains the .DO. It may be part of a macro expansion defined in a file assembled prior to the source containing the .DO, or it may be part of a macro expansion defined in MASM.PS (or MASM16.PS). It may not be in the next file to be assembled. The following code illustrates this:

```
File A:
   .
   .
.DO     5
.BLK    1
.EOF
```

```
File B:
1
2
.ENDC
   .
   .
.END
```

The command

$$)\text{XEQ} \left\{ \begin{array}{l} \text{MASM} \\ \\ \text{MASM16} \end{array} \right\} \quad \text{A□B<nl>}$$

will cause an error because the .DO in file A cannot be terminated by an .ENDC in file B.

Examples

Source code for the first example:

```
        .DO 3    ;ASSEMBLE THE FOLLOWING CODE THREE TIMES.
        10
        20
        .ENDC    ;END OF '.DO' LOOP
```

Assembly listing for this code:

```
08        000003      .DO 3    ;ASSEMBLE THE FOLLOWING CODE THREE TIMES.
09 00000'000010       10
10 00001'000020       20
11                    .ENDC    ;END OF .DO LOOP
12 00002'000010       10
13 00003'000020       20
14                    .ENDC    ;END OF .DO LOOP
15 00004'000010       10
16 00005'000020       20
17                    .ENDC    ;END OF .DO LOOP
```

.DO (continued)

The second example shows how to use the .DO pseudo-op to perform conditional assembly. The source code for this example is:

```
        1

        A=3
        .DO A==3        ;ASSEMBLE THE FOLLOWING CODE ONCE
        2               ;IF THE VALUE OF  A=3.  OTHERWISE,
        .ENDC           ;DO NOT ASSEMBLE THE CODE AT ALL.

        3
        .END
```

The assembly listing for this code is:

```
08 00000'000001      1
09
10       000003      A=3
11       000001      .DO A==3        ;ASSEMBLE THE FOLLOWING CODE ONCE
12 00001'000002      2               ;IF THE VALUE OF  A=3.  OTHERWISE,
13                   .ENDC           ;DO NOT ASSEMBLE THE CODE AT ALL.
14
15 00002'000003      3
16                   .END
```

References

"Absolute Expressions" - Chapter 3
"Loops and Conditionals in Macros" - Chapter 5
"Repetitive and Conditional Assembly" - Chapter 6

.DUSR

Define a user symbol for cross-referencing.

Syntax

```
                          ( instruction )
.DUSR□user-symbol=       {               }
                          ( expression  )
```

Purpose

The .DUSR pseudo-op defines **user-symbol** as having the value of **expression or instruction.**

Expression may be any legal Macroassembler expression. **Instruction** may be any legal 16-bit ECLIPSE assembly language instruction. If you supply an instruction, MASM computes the assembled value of that instruction and assigns it to **user-symbol.** MASM pads or truncates the instruction's value to produce a single-precision (16-bit) integer. Refer to "Assignments" in Chapter 2 for more information about using instructions in assignments.

Once defined, you may use **user-symbol** anywhere you would use a single-precision (16-bit) operand. In addition, you may change the value of **user-symbol** at any time by using .DUSR (assuming that you do not use the /M function switch (see Chapter 8)).

The above description makes it clear that the .DUSR pseudo-op performs the same function as the simple assignment statement (see Chapter 2). For example, the statements in the two columns of Table 7-2 assign equivalent values to the symbols A, B, C, and D.

The only difference between issuing a simple assignment statement and using the .DUSR pseudo-op concerns the cross-reference listing. The Macroassembler considers .DUSR symbols as instruction symbols for the purpose of the cross-reference. Thus, by default, the cross-reference does not contain .DUSR symbols but does contain symbols that appear in simple assignment statements.

MASM treats the I/O device symbols (e.g., LPT, TTO, PTR), the ALC skip mnemonics (e.g., SKP, SNC, SNR), and the hardware stack location mnemonics (e.g., SP, FP, SL, SFA) as if they were defined by the .DUSR pseudo-op.

Table 7-2. DUSR Assignments Versus Simple Assignments

.DUSR Assignments	Simple Assignments
.DUSR A=10	A=10
.DUSR B=A+20	B=A+20
.DUSR C=XWLDA 0,0	C=XWLDA 0,0
.DUSR D=.RDX	D=.RDX

Example

```
                        .TITLE ASSGN
06                      .NREL
07      000002          A=2               ;'A,' 'B,' AND 'C' ARE USER SYMBOLS.
08      000020          .DUSR  B=20
09      000030          .DUSR  C=30
10      000070          A=70              ;YOU MAY REDEFINE 'A' WITH
11                                        ;AN ASSIGNMENT STATEMENT.
12      000003          .DUSR  B=3        ;YOU MAY REDEFINE 'B' WITH '.DUSR.'
13                      .END
```

The cross-reference for this code is:

```
   A     000070        1/07#   1/10#
```

By default, MASM does not include B and C in the cross-reference listing because they are defined by .DUSR.

NOTE: Symbols defined by assignment statements cannot be redefined by .DUSR statements and vice versa.

References

"Assignments" - Chapter 2
"Cross-Reference Listing" - Chapter 4
"Expressions" - Chapter 2
"Symbols" - Chapter 2
"User Symbols" - Chapter 2

.DXOP

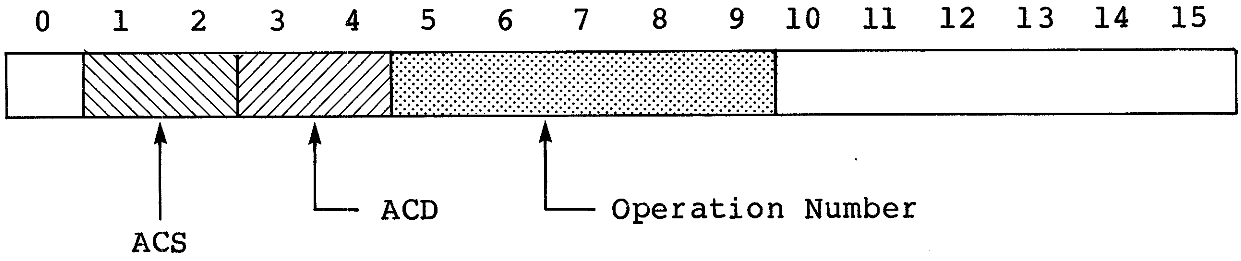
Define an instruction with source, destination, and operation fields.

Syntax

$$\textbf{.DXOP}\square\textbf{user-symbol} = \begin{Bmatrix}\textbf{instruction}\\ \textbf{expression}\end{Bmatrix}$$

Purpose

This pseudo-op defines **user-symbol** as a reference symbol with source and destination fields and an operation number field. The symbol has the value of **instruction** or **expression.** This symbol implies the format of an instruction that requires a source and a destination accumulator. Three fields are required and are assembled as shown.



The format for using this symbol is:

user-symbol□source-ac□destination-ac□operation-number

NOTE: You should not use the .DXOP pseudo-op in programs that will run on the ECLIPSE MV/Family line of computers. The operation code number generated by the .DXOP pseudo-op is not valid on ECLIPSE MV/Family computers. See the <u>Principles of Operation--32-Bit ECLIPSE® Systems</u> manual for more information.

.DXOP (continued)

Example

```
08 00000'100030      XOP 0,0,0        ;EXTENDED OPERATION INSTRUCTION
09
10       100030      .DXOP POX=100030  ;DEFINE POX AS XOP
11
12 00001'100030      POX 0,0,0        ;NEWLY DEFINED INSTRUCTION
13
14
15 00002'155230      XOP 2,3,12       ;EXECUTE EXTENDED OPERATION 12 USING
16                                    ;ACCUMULATORS 2 AND 3
17
18 00003'155230      POX 2,3,12       ;EXECUTE EXTENDED OPERATION 12 USING
19                                    ;ACCUMULATORS 2 AND 3
```

References

"Assembly Language Instructions" - Chapter 2
"Symbol Table Pseudo-Ops" - Chapter 6

.EJEC

Begin a new listing page.

Syntax

.EJEC

Purpose

The .EJEC pseudo-op directs the assembler to begin a new page in the assembly listing output (after listing the .EJEC source statement).

Example

The source code for this example is:

```
          .
          .
          .
        MOV 0,1
        .EJEC           ;START A NEW LISTING PAGE
        LDA 2,0,1
        MOV 2,3
          .
          .
```

The assembly listing for this code is:

```
 0001  .MAIN    AOS ASSEMBLER REV 03.40         11:37:49 07/19/81
                           .
                           .
                           .
08 00042 105000          MOV 0,1
09                       .EJEC          ;START A NEW LISTING PAGE


<page break>

 0002  .MAIN
01 00043 030400          LDA 2,0,1
02 00044 155000          MOV 2,3
                           .
                           .
                           .
```

.EJEC (continued)

Reference

"Assembly Listing" - Chapter 4

.END

End-of-program indicator.

Syntax

.END<□expr>

Purpose

Use the .END pseudo-op to terminate your source program. The assembler does not process any source code that follows the .END pseudo-op; so, this should be the last statement in your source.

If you assemble several modules at once, only the last one should include an .END statement (use .EOF to end the other modules). If you do not include an .END pseudo-op at the end of the last module on the assembly command line, MASM supplies one for you (without an argument).

The optional expr argument specifies a starting address for execution of your program file. You must supply a start execution address in at least one of your source modules or Link returns an error.

Example

```
                        .TITLE MOD1
06                      .NREL
07 00000'102400 START:  SUB  0,0
                          .
                          .
                          .
                                       ;END OF MODULE 'MOD1.' BEGIN EXECUTION
13                      .END START      ;OF PROGRAM AT LOCATION 'START.'
```

References

"Expressions" - Chapter 2
AOS Link User's Manual or
AOS/VS Link and Library File Editor User's Manual
"File Termination" - Chapter 6

.ENDC

End of conditional or repetitive assembly.

Syntax

.ENDC<□user-symbol>

Purpose

The .ENDC pseudo-op terminates source lines whose assembly is repetitive (lines following .DO) or conditional (lines following .IFE, .IFG, .IFL,or .IFN).

If user-symbol is used, the .ENDC pseudo-op terminates assembly of lines following the .DO or .IFx and suppresses the assembly of lines following .ENDC until the scan encounters another user-symbol enclosed in square brackets.

NOTE: Assembly is suppressed only when the source lines terminated by .ENDC are assembled; if the source lines are not assembled, assembly continues immediately following .ENDC.

Example

```
08        000001         A=1
09        000001         .DO A==1      ;MASM ASSEMBLES THESE
10 00000'000010          10            ;LINES BECAUSE THE '.DO'
11 00001'000020          20            ;CONDITION IS TRUE.
12                       .ENDC SKIP
13                       30            ;MASM SUPPRESSES ASSEMBLY OF LINES
14                       40            ;UNTIL IT FINDS 'SKIP.'
15 00002'000050 [SKIP]   50
16 00003'000060          60
17 00004'000070          70
18        000000         .DO A==2      ;MASM DOES NOT ASSEMBLE
19                       60            ;THESE LINES BECAUSE THE '.DO'
20                       70            ;CONDITION IS FALSE.
21                       .ENDC PIKS
22 00005'000100          100           ;MASM ASSEMBLES THESE LINES BECAUSE THE
23 00006'000200          200           ;PRECEDING '.DO' CONDITION WAS FALSE.
24 00007'000300 [PIKS]   300
25                       .END
```

References

.DO and .IF pseudo-op descriptions - Chapter 7
"Repetitive and Conditional Assembly" - Chapter 6

.ENT

Define an external entry.

Syntax

.ENT□user-symbol...

Purpose

The .ENT pseudo-op declares **user-symbol** as a symbol that you define in this source module but that you may reference from separately assembled modules.

You must define a **user-symbol** in the module containing the .ENT declaration (see "User Symbols" in Chapter 2). This **user symbol** must be unique among all external symbols you define in the modules you intend to link together. If the symbol is not unique, Link issues a message indicating multiply defined entries.

To reference **user-symbol** from a separately assembled module, use one of the following pseudo-ops:

.EXTD .EXTN .EXTU

Example

<u>First module:</u>

```
        .TITLE MODA        ;MODULE 'MODA'
        .ENT PNTR          ;'PNTR' IS DEFINED IN THIS MODULE AND
                           ;MAY BE REFERENCED IN OTHER MODULES.

        .ZREL
PNTR:   TABLE
        .NREL
TABLE:  0
        1
        2
        3
        4

        .END
```

Second module:

```
.TITLE MODB     ;SEPARATELY ASSEMBLED MODULE 'MODB'
        .EXTD PNTR      ;'PNTR' IS DEFINED IN ANOTHER MODULE.
        .NREL
        LDA 0,@PNTR     ;REFERENCE IS TO EXTERNALLY DEFINED
                        ;SYMBOL 'PNTR'
            .
            .
            .
        .END
```

References

"Intermodule Communication" - Chapter 6
"User Symbols" - Chapter 2

.ENTO

Define an overlay entry.

Syntax

.ENTO□user-symbol

Purpose

.ENTO is used when a program is to become an overlay within an overlay segment. The pseudo-op associates **user-symbol** with the node number and overlay number of a particular overlay. The overlay may then be referenced from another program by **user-symbol. User-symbol** must be declared as an .EXTN in the referencing program. Caution: **user-symbol** cannot appear elsewhere in the program in which it is declared as the name of an overlay, because its value is assigned at link time.

Both shared and unshared code can be placed in the same assembly module. If this module is placed in an overlay, code in the module will actually be split into two different overlays. The shared code will go to an overlay in the shared area; unshared code will become an overlay in the unshared area. To identify the two overlays, two .ENTOs must be used. Each .ENTO must follow immediately its associated .NREL statement.

Examples

First module:

```
        .TITLE OVERLAY
        .ENTO ENTRY     ;OVERLAY NAME
             .                                        Overlay
             .
             .
        .END


        .TITLE PROG
        .EXTN ENTRY   ;OVERLAY NAME DEFINED
             .        ;AS EXTERNAL
             .
             .
ENADR:  ENTRY
             .                                       Referencing program
             .
        LDA 0,ENADR ;LOAD OVERLAY NAME IN AC0
        ?OVLOD      ;SYSTEM CALL TO LOAD
                    ;OVERLAY
             .
             .
        END
```

Second module:

```
        .TITLE OVRLY
        .NREL 0
        .ENTO EO1         ;ENTRY POINT FOR
                          ;UNSHARED CODE OVERLAY
             .
             .
             .
        .NREL 1
        .ENTO EO2         ;ENTRY POINT FOR
                          ;SHARED CODE OVERLAY
             .
             .
        .END
```

Reference

"Intermodule Communication" - Chapter 6

.EOF

Explicit end-of-file.

Syntax

```
.EOF
```

Purpose

The .EOF pseudo-op provides the assembler with an explicit end-of-file indicator. This pseudo-op indicates the end of one source module, but implies that more source modules follow in the current assembly. Thus, use .EOF to terminate each source module in the MASM command line, except the last one (the last module should end with .END).

If you do not include .EOF pseudo-ops in your source modules, MASM automatically supplies them for you.

Example

<u>First module:</u>

```
        .TITLE MODA      ;THE FIRST PIECE OF SOURCE CODE
                         ;RESIDES IN FILE 'MODA.'
        .NREL
START:  1
        2
        3
        4
        .EOF             ;END OF FILE 'MODA'
                         ;BUT NOT OF SOURCE CODE
```

Second module:

```
.TITLE MODB     ;THE SECOND PART OF THE SOURCE CODE
                       ;IS IN FILE 'MODB.'

        .NREL
        1
        2
        3
        .END  START     ;END OF CURRENT ASSEMBLY.  START
                       ;PROGRAM EXECUTION AT LOCATION 'START.'
```

The MASM command line that assembles these two source modules is:

```
       ( MASM   )
)XEQ  {          }  A□B<nl>
       ( MASM16 )
```

Reference

"File Termination" - Chapter 6

.EXTD

Define an external displacement reference: 8 bits.

Syntax

.EXTD□user-symbol...

Purpose

This pseudo-op declares **user-symbol** as a symbol that you reference in this source module but that you define in a separately assembled module. You must declare **user-symbol** with an .ENT pseudo-op in the module that defines it. Also, you cannot redefine **user-symbol** within the current assembly.

When you use an .EXTD symbol in your module, you must make sure the corresponding field is at least 8 bits wide. If you use an .EXTD symbol in a field that is less than 8 bits wide, MASM returns an error.

You may use an .EXTD symbol as a displacement or lower page zero (ZREL) address in any memory reference instruction (MRI). When used in this manner, the symbol's value must satisfy one of the following equations:

$0 <= \text{lower page zero address} <= 377_8$

$-200_8 <= \text{displacement} <= 177_8$

Be sure the value of the .EXTD symbol can fit into the corresponding field at link time. Otherwise, you will receive a Link error.

Example

First module:

```
        .TITLE MODA     ;MODULE 'MODA'
        .ENT D          ;'D' IS DEFINED IN THIS MODULE AND
                        ;MAY BE REFERENCED IN OTHER MODULES

        .ZREL
D:      TABLE           ;'D' IS A LABEL IN PAGE ZERO RELOCATABLE
                        ;MEMORY AND THEREFORE ITS VALUE CAN BE
                        ;EXPRESSED IN 8 BITS.
        .NREL
TABLE:  0
        1
        2
        3
        4

        .END
```

Second module:

```
.TITLE MODB     ;SEPARATELY ASSEMBLED MODULE 'MODB'
        .EXTD D         ;'D' IS DEFINED EXTERNALLY AND CAN
                        ;BE EXPRESSED IN AN 8-BIT FIELD.
        .NREL
        LDA 0,D         ;LOAD THE VALUE AT LOCATION 'D'
                        ;INTO AC0.
           .
           .
           .
        .END
```

References

"Intermodule Communication" - Chapter 6
"User Symbols" - Chapter 2

.EXTN

Define an external normal reference.

Syntax

.EXTN□user-symbol...

Purpose

This pseudo-op declares **user-symbol** as a symbol that you reference in this source module but that you define in a separately assembled module. You must declare **user-symbol** with an .ENT pseudo-op in the module that defines it. Also you cannot redefine **user-symbol** within the current assembly.

When you use an .EXTN symbol in your program, make sure the corresponding field is at least 16 bits wide. If you use an .EXTN symbol in a field that is less than 16 bits wide, MASM returns an error.

An .EXTN **user-symbol** specifies only the contents of a 16-bit storage word. The value at bind time must therefore be a number in the range 0 through $2^{16}-1$.

Example

```
First module:

        .TITLE A          ;SOURCE MODULE 'A'
        .ENT N
        .NREL
        N=3777            ;THE VALUE OF 'N' CAN BE EXPRESSED IN 16 BITS.
          .
          .
          .
        .END
```

Second module:

```
.TITLE B        ;SEPARATELY ASSEMBLED MODULE 'B'
        .EXTN N         ;'N' IS DEFINED IN ANOTHER MODULE AND
                        ;ITS VALUE MAY BE EXPRESSED IN 16 BITS.
        .NREL
        ADDI N,0        ;ADD THE IMMEDIATE VALUE 'N' (16 BITS)
                        ;TO THE CONTENTS OF AC0.
           .
           .
           .
        .END
```

References

"Intermodule Communication" - Chapter 6
"User Symbols" - Chapter 2

.EXTU

Treat undefined symbols as external displacements.

Syntax

.EXTU

Purpose

This pseudo-op causes the assembler to treat all symbols that are undefined after pass 1 as if they had appeared in an .EXTD statement. In other words, MASM assumes that all undefined symbols will be defined in other modules at link time.

NOTE: In general, you should not use this pseudo-op. .EXTU puts off errors until Link time and increases assembler overhead.

Example

```
                     .TITLE  PROG
                     .EXTU
00000 024000$        LDA 1,SYMB      ;SYMB IS NOT A DEFINED
                                     ;SYMBOL.  EXTU CAUSES IT
                                     ;TO BE DEFINED AS AN EXTERNAL
                                     ;DISPLACEMENT
                     .END
```

Reference

"Intermodule Communication" - Chapter 6

.FORC

Force-link a module from a library.

Syntax

.FORC

Purpose

The .FORC pseudo-op directs Link to unconditionally link this object file from a library into your program file.

Normally, Link includes an object module from a library only if that file satisfies an external reference appearing in another module. However, if you use .FORC in a module that resides in a library, that module will be linked whenever the library name occurs in a Link command line. You might use .FORC for a module you plan to build into a library.

Example

```
        .TITLE SQUARE   ;'SQUARE' IS PART OF A LIBRARY.  WHENEVER THE
        .FORC           ;LIBRARY NAME APPEARS IN A LINK COMMAND LINE,
        .ENT CUBE       ;MODULE 'SQUARE' WILL BE LINKED INTO THE PROGRAM
        .NREL           ;FILE.  THUS THE PROGRAM WILL HAVE ACCESS TO
                        ;ENTRY 'CUBE,' EVEN IF THIS VERSION OF THE
                        ;PROGRAM DOESN'T REFERENCE 'CUBE.'
CUBE:   1
        10
        33
        100
        .END
```

References

AOS Link User's Manual or
AOS/VS Link and Library File Editor User's Manual
AOS Library File Editor User's Manual

.GADD

Add expression value to symbol.

Syntax

.GADD□user-symbol□expression

Purpose

This pseudo-op generates a storage word whose contents are resolved at link time. The Macroassembler searches for the value of **user-symbol.** If found, its value is added to **expression to** form the contents of the storage word. If the **user-symbol** is not found, a Link error results and the storage word will contain just the value of **expression.**

User-symbol must be a symbol defined in some separately assembled program and appearing in that program in an .ENT, .ENTO, .PENT, or .COMM pseudo-op.

Example

```
    .TITLE PROG1                                 )
        .                                        |
        .                                        |
    .ENT A                                       } PROGRAM 1
    .LOC 200                                     |
A:                      ;VALUE OF A IS 200       )
    .END

    .TITLE PROG2                                 )
        .                                        |
        .                                        |
    .EXTN A             ;A IS EXTERNAL           } PROGRAM 2
    .GADD A, 3+4        ;3+4 ADDED TO 'A'        |
                        ;AT LINK TIME            |
    .END                                         )
```

Reference

"Intermodule Communication" - Chapter 6

.GLOC

Initialize data fields relative to an external symbol.

Syntax

.GLOC□external-symbol

Purpose

The .GLOC pseudo-op defines a block of data whose starting address will equal the value of **external-symbol** at link time. Define your data immediately after the .GLOC statement. The first occurrence of a .LOC, .NREL, .ZREL, or .END pseudo-op terminates the data block. Another .GLOC statement also ends the current data block.

To reference the data block from a separately assembled module, include **external-symbol** in an .ENT or .COMM statement within that module. In the current module, you must declare **external-symbol** as externally defined with an .EXTD, .EXTN, or .EXTU pseudo-op.

You may not include external references, label definitions, or label references within the .GLOC data block. All other expressions are legal.

By using .GLOC, you may initialize your data fields at link time instead of at runtime. This can save memory space and reduce your program's execution time.

.GLOC (continued)

Example

```
First module:

        .TITLE A          ;SOURCE MODULE 'A'
        .NREL
        .COMM DATA, 4     ;COMMON AREA 'DATA' CONTAINS
                          ;FOUR WORDS OF MEMORY.
          .
          .
          .
        .END


Second module:

.TITLE B          ;SEPARATELY ASSEMBLED MODULE 'B'
        .EXTN DATA
        .GLOC DATA        ;THIS STATEMENT DIRECTS LINK TO
        1                 ;INITIALIZE COMMON AREA 'DATA' WITH
        2                 ;THE VALUES 1 THROUGH 4.
        3
        4
        .END
```

Reference

"Intermodule Communication" - Chapter 6

.GOTO

Suppress assembly temporarily.

Syntax

.GOTO□user-symbol

Purpose

This pseudo-op suppresses the assembly of lines until the Macroassembler encounters another **user-symbol** enclosed in square brackets.

Example

```
07                          .NREL
08 00000'021001             LDA 0,1,2        ;ASSEMBLED INSTRUCTION
09 00001'050001             STA 2,1,0        ;ASSEMBLED INSTRUCTION
10
11                          .GOTO LABEL      ;'GOTO' PSEUDO-OP
12
13                          LDA 1,1,1        ;UNASSEMBLED INSTRUCTION
14                          STA 2,2,2        ;UNASSEMBLED INSTRUCTION
15
16 00002'024401 [LABEL] LDA 1,1,1        ;ASSEMBLY RESUMED
17
18                          .END
```

Reference

"Repetitive and Conditional Assembly" - Chapter 6

.GREF

Add expression value to symbol.

Syntax

.GREF□user-symbol□expression

Purpose

The .GREF pseudo-op is similar to the .GADD pseudo-op. It generates a storage word whose contents are resolved at link time. The value of **user-symbol is** searched for and, if found, is added to **expression.** However, unlike .GADD, a carry from the least-significant 15 bits does not alter bit zero. If **user-symbol** is not found, a Link error results, and the storage word will contain just the value of **expression.**

Example

```
        .TITLE PROG1                            )
            .                                   |
            .                                   |
        .ENT A                                  }  PROGRAM 1
        .LOC 200                                |
    A:                    ;VALUE OF A IS 200    |
        .END                                    )

        .TITLE PROG2                            )
            .                                   |
            .                                   |
        .EXTN A           ;A IS EXTERNAL        |
            .                                   }  PROGRAM 2
        .GREF A, 3+4      ;3+4 ADDED TO 'A'     |
                          ;AT LINK TIME         |
                                                |
        .END                                    )
```

Reference

"Intermodule Communication" - Chapter 6

.IFE, .IFG, .IFL, .IFN

Perform conditional assembly.

Syntax

.IFE□abs-expr
.IFG□abs-expr
.IFL□abs-expr
.IFN□abs-expr

Purpose

These pseudo-ops direct MASM to either assemble or bypass portions of your module on the basis of **abs-expr.** The Macroassembler assembles the source lines following an .IF pseudo-op if the value of **abs-expr** satisfies the condition defined by that pseudo-op. If the value of **abs-expr** does <u>not</u> satisfy that condition, MASM resumes assembly after the .ENDC pseudo-op.

Abs-expr must be an absolute expression.

The four .IF pseudo-ops define the following conditions:

.IFE□abs-expr	Assemble if **abs-expr** equals 0
.IFG□abs-expr	Assemble if **abs-expr** is greater than 0
.IFL□asb-expr	Assemble if **abs-expr** is less than 0
.IFN□abs-expr	Assemble if **abs-expr** does not equal 0

You must terminate the conditional assembly lines with the .ENDC pseudo-op. Thus, the conditional portion of your module has the general form:

```
.IFx□abs-expr     ;One of the four conditional pseudo-ops
.                 ;(.IFE, .IFG, .IFL, or .IFN).
.
.                 ;Assemble these source lines if abs-expr
.                 ;satisfies the .IFx condition.
.
.ENDC             ;Terminate conditional assembly.
```

You may nest conditional pseudo-ops to any reasonable depth (at least $3{,}000_{10}$ levels). When nesting .IFs, be sure the innermost .IF corresponds to the innermost .ENDC, etc.

Note that each .IF condition is a form of a .DO statement. For example, the statement .IFE A is equivalent to .DO A==0. Both

.IFE, .IFG, .IFL, .IFN (continued)

direct MASM to assemble the following code once if A equals 0.

In the value field of the assembly listing, the assembler places a 1 if **abs-expr** satisfies the pseudo-op condition and 0 if it does not satisfy the condition.

Example

```
08        000002     A=2
09        000003     B=3
10        000000     .IFE B-A        ;THESE LINES DO NOT ASSEMBLE
11                   100             ;BECAUSE 'B-A' IS NOT EQUAL
12                   200             ;TO 0.
13                   .ENDC
14
15        000001     .IFG B-A        ;THESE LINES ASSEMBLE
16 00000'000100      100             ;BECAUSE 'B-A' IS GREATER
17 00001'000200      200             ;THAN 0.
18                   .ENDC
19
20        000000     .IFL B-A        ;THESE LINES DO NOT ASSEMBLE
21                   100             ;BECAUSE 'B-A' IS NOT LESS
22                   200             ;THAN ZERO.
23                   .ENDC
24
25        000001     .IFN B-A        ;THESE LINES ASSEMBLE
26 00002'000100      100             ;BECAUSE 'B-A' IS NOT
27 00003'000200      200             ;EQUAL TO 0.
28                   .ENDC
```

Reference

"Absolute Expressions" - Chapter 3
"Loops and Conditionals in Macros" - Chapter 5
"Repetitive and Conditional Assembly" - Chapter 6

.LMIT

Bind part of an object module.

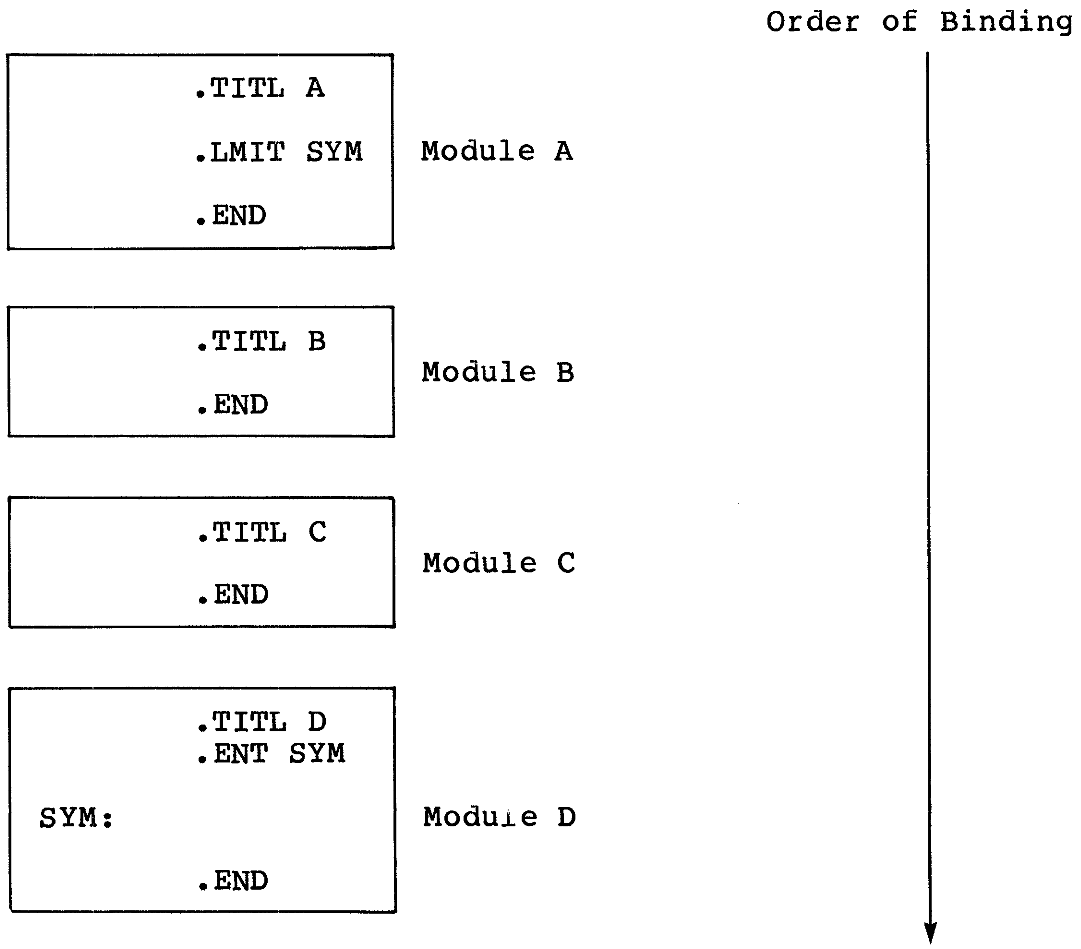
Syntax

.LMIT□user-symbol

Purpose

This pseudo-op specifies partial binding of an object file. A .LMIT pseudo-op in one OB file will cause an OB later in the binding process to be partially bound. **User-symbol** is an entry point appearing in an OB file that stops binding when the symbol is encountered.

Examples



```
        .TITL A

        .LMIT SYM

        .END
```

```
        .TITL B

        .END
```

```
        .TITL C

        .END
```

```
        .TITL D
        .ENT SYM

SYM:

        .END
```

.LMIT (continued)

In the example, Module D contains the entry point SYM that corresponds to the **user-symbol** SYM appearing in the .LMIT pseudo-op in Module A. Link will bind D up to, but not including, the line identified by SYM.

The limiting symbol (in this case SYM) must be declared an entry point in the module to be partially bound. If the limiting symbol is NREL, all of Module D ZREL will be bound and Module D NREL will be bound up to the limiting symbol. If the limiting symbol is in ZREL, NREL will be completely bouna and ZREL will be bound up to the limiting symbol. A module may be limited in NREL and in ZREL by two different symbols. If two symbols limit either NREL or ZREL, the lower symbol in value will be the limiting symbol. There are no restrictions on the number of limiting symbols that may be used.

If the limited module is in a library, the module will be bound up to its limiting symbol. This is so even if the module would otherwise not have been bound (i.e., even if there is no undefined external to cause the library to be bound).

If there is an undefined external that references an entry point in the unbound part of the module, the module will still be only partially bound as indicated by the limiting symbol.

All of the entry points of a partially bound module will appear on the load map as though the corresponding parts of the module were actually bound. Any references to them will be resolved, but will actually point into the succeeding module.

Reference

"Miscellaneous Pseudo-Ops" - Chapter 6

.LOC

Set the current location counter.

Syntax

.LOC<□expr>

Purpose

The .LOC pseudo-op sets the current location counter to the value and relocation base given by *expr*. The location counter is an assembler variable that holds the address of the next memory location MASM will assign.

The argument you supply to .LOC may be any legal assembler expression. If you do not supply an argument, MASM returns an error.

As an example, if *expr* resolves to an absolute value, then the assembler sets the current location counter to that value and subsequent addresses are not relocatable (they are absolute).

Value

You may use .LOC as a value symbol, in which case it has the value and relocation property of the current location counter.

The exception to this is that when .LOC is placed on the assembler stack, only the relocation property is saved. For example, using .PUSH and .POP, you may save and restore the relocation base of the location counter as follows:

```
.PUSH  .LOC ;Save the relocation base
         .  ;of the location counter on the stack.
         .
         .
.LOC   .POP ;Set the relocation base of the location
            ;counter equal to the entry on the top of the stack.
```

You can save the current relocation base within a macro and restore it correctly without affecting the relative location counter value, which may have been altered within the macro.

.LOC (continued)

Example

```
06                      .NREL         ;UNSHARED NREL CODE RELOCATION
07 00000'000001 N:      1
08 00001'000002         2
09 00002'000003         3
10
11       000100         .LOC 100      ;SET THE LOCATION COUNTER TO
12 00100 000004         4             ;ABSOLUTE LOCATION 100.
13 00101 000005         5
14 00102 000006         6
15
16       000050'        .LOC N+50     ;SET THE LOCATION COUNTER TO THE
17 00050'000007         7             ;SAME RELOCATION BASE AS N (UNSHARED
18 00051'000010         10            ;NREL) AND START ASSIGNING LOCATIONS
19 00052'000011         11            ;AT THE 50TH ADDRESS AFTER N.
```

References

"Assigning Locations" - Chapter 3
"Expressions" - Chapter 2
"Location Counter" - Chapter 3

.LPOOL

Dump the currently defined literals.

Syntax

.LPOOL

Purpose

This pseudo-op directs the Macroassembler to dump the currently defined literals into a data block. The block is dumped at the value of the current partition's location counter. The .LPOOL pseudo-op will be flagged with a literal (L) error if it is used without the .NLIT pseudo-op.

Both shared and unshared .NREL literals may be used in the same program. However, the literals must be dumped in their respective partitions to avoid addressing errors.

Within a literal pool, (i.e., the data block dumped by the .LPOOL), all literals that can be optimized will be. Literals that cannot be optimized are forward reference expressions and external references. The literals are dumped in the order in which they were created.

NOTE: The programmer must ensure that the data block is not executed.

.LPOOL (continued)

Example

```
                           .TITLE MOD1
06                         .ENT ENT1
07                         .ENT ENT2
08                         .EXTD EXIT
09                         .NLIT          ;ASSIGN LITERALS TO NREL
10        000001           .NREL 1        ;SHARED NREL CODE
11 00000!132400 ENT1:     SUB 1,2
12 00001!157000           ADD 2,3
13 00002!020403 OUT:      LDA 0,=12      ;LITERAL '12' IN LDA INSTRUCTION.
14 00003!024403           LDA 1,=34      ;LITERAL '34' IN LDA INSTRUCTION.
15 00004!000000$          JMP EXIT
16 00005!000012           .LPOOL         ;DUMP LITERALS '12' AND '34'
17        000034
18
19        000000           .NREL 0        ;UNSHARED NREL CODE
20 00000'030403 ENT2:     LDA 2,=21      ;LITERAL '21' IN LDA INSTRUCTION.
21 00001'034403           LDA 3,=54      ;LITERAL '54' IN LDA INSTRUCTION.
22 00002'002403           JMP @=OUT      ;JUMP INDIRECTLY TO OUT.
23 00003'000021           .LPOOL         ;DUMP LITERALS '21,' '54,' AND 'OUT.'
24        000054
25        000002!
26
27                         .END
```

Reference

"Using Literals in Memory Reference Instructions" - Chapter 3

.MACRO

Define a macro.

Syntax

.MACRO□user-symbol
macro-definition-string

Purpose

The .MACRO pseudo-op defines **user-symbol** as the name of **macro-definition-string. Macro-definition-string** is one or more source lines that you use repeatedly in your module. After defining the macro, you simply insert **user-symbol** in your source module, and the assembler substitutes **macro-definition-string.**

When defining a macro, you must terminate the **macro definition string** with the percent character **(%).** We recommend that % be the only character on the source line.

Examples

The source code for this example is:

```
        .NREL
        .MACRO TEST      ;DEFINE MACRO 'TEST.'
        ^1               ;MACRO DEFINITION CONSISTS OF 3
        ^2               ;DATA STATEMENTS THAT GET THEIR
        ^3               ;VALUES FROM THE FIRST 3 ARGUMENTS
        %                ;PASSED TO 'TEST.'

        TEST  4,5,6      ;CALL 'TEST' WITH THREE ARGUMENTS

        TEST  0,1,2      ;CALL 'TEST' WITH THREE MORE ARGUMENTS
```

.MACRO (continued)

```
The assembly listing for this example is:

06                    .NREL
07                    .MACRO TEST      ;DEFINE MACRO 'TEST.'
08                    ^1               ;MACRO DEFINITION CONSISTS OF 3
09                    ^2               ;DATA STATEMENTS THAT GET THEIR
10                    ^3               ;VALUES FROM THE FIRST 3 ARGUMENTS
11                    %                ;PASSED TO 'TEST.'
12
13                    TEST  4,5,6      ;CALL 'TEST' WITH THREE ARGUMENTS
14 00000'000004       4               ;MACRO DEFINITION CONSISTS OF 3
15 00001'000005       5               ;DATA STATEMENTS THAT GET THEIR
16 00002'000006       6               ;VALUES FROM THE FIRST 3 ARGUMENTS
17
18                    TEST  0,1,2      ;CALL 'TEST' WITH THREE MORE ARGUMENTS
19 00003'000000       0               ;MACRO DEFINITION CONSISTS OF 3
20 00004'000001       1               ;DATA STATEMENTS THAT GET THEIR
21 00005'000002       2               ;VALUES FROM THE FIRST 3 ARGUMENTS
```

References

"Macros" - Chapter 5
"User Symbols" - Chapter 2

.MCALL

Indicate macro usage.

Value

The .MCALL pseudo-op is a value symbol. .MCALL has the value 1 if the macro containing it was called previously on this assembly pass, and the value 0 if this is the first call to that macro on the current pass. Thus, you generally use .MCALL when you want the assembler to use one macro expansion the first time a macro is called and a different expansion for subsequent calls to that macro.

If you use .MCALL outside a macro, its value is -1.

Example

```
06                      .ZREL
07 00000-000100         TABL1:   .BLK  100
08 00100-000100         TABL2:   .BLK  100
09 00200-000000-        LOC1: TABL1
10 00201-000100-        LOC2: TABL2
11
12                      .NREL
13
14                      .MACRO MC          ;DEFINE MACRO 'MC.'
15                      .IFE   .MCALL      ;ON THE FIRST CALL TO MACRO 'MC'
16                      LDA    0,LOC1      ;ASSEMBLE THE FIRST 'LDA' INSTRUCTION.
17                      .ENDC SKIP         ;END OF CONDITIONAL
18                      LDA    0,LOC2      ;ON SUBSEQUENT CALLS TO 'MC'
19                      [SKIP]             ;ASSEMBLE THE SECOND 'LDA' INSTRUCTION.
20                      %
21
22                      MC                 ;FIRST CALL TO 'MC' ('.MCALL' EQUALS 0)
23         000001       .IFE   .MCALL      ;ON THE FIRST CALL TO MACRO 'MC'
24 00000'020200-        LDA    0,LOC1      ;ASSEMBLE THE FIRST 'LDA' INSTRUCTION.
25                      .ENDC SKIP         ;END OF CONDITIONAL
26                      LDA    0,LOC2      ;ON SUBSEQUENT CALLS TO 'MC'
27                      [SKIP]             ;ASSEMBLE THE SECOND 'LDA' INSTRUCTION.
28
29                      MC                 ;SECOND CALL TO 'MC' ('.MCALL' EQUALS 1)
30         000000       .IFE   .MCALL      ;ON THE FIRST CALL TO MACRO 'MC'
31                      LDA    0,LOC1      ;ASSEMBLE THE FIRST 'LDA' INSTRUCTION.
32                      .ENDC SKIP         ;END OF CONDITIONAL
33 00001'020201-        LDA    0,LOC2      ;ON SUBSEQUENT CALLS TO 'MC'
34                      [SKIP]             ;ASSEMBLE THE SECOND 'LDA' INSTRUCTION.
```

.MCALL (continued)

References

"Macro-Related Pseudo-Ops" - Chapter 5
"Macros" - Chapter 5

.NLIT

Assign literals to .NREL.

Syntax

.NLIT

Purpose

This pseudo-op tells the Macroassembler to assign literals to NREL instead of ZREL. The .LPOOL pseudo-op must be used to dump the literals into a data block within the relative range of the literal references. The .NLIT pseudo-op must occur before any literals are generated. If it does not, the .NLIT will be flagged with a literal (L) error and will be ignored (i.e., the literals will continue to be assigned .ZREL locations).

Example

See the .LPOOL pseudo-op description for an example.

Reference

"Using Literals in Memory Reference Instructions" - Chapter 3

.NOCON

Inhibit or re-enable the listing of conditional lines.

Syntax

.NOCON□abs-expr

Purpose

The .NOCON pseudo-op either inhibits or permits the listing of the conditional portions of the source module that do not meet the conditions given for assembly. That is, .NOCON either inhibits or enables the listing of false conditionals. If the value of **abs-expr** does not equal zero, the assembler inhibits listing; if the value of **abs-expr** equals zero, the assembler lists all conditionals. **Abs-expr** must be an absolute expression.

.NOCON does not affect the conditional portions of the source program that MASM assembles. Again, this pseudo-op influences only the listing of conditionals that are false (those .DOs and .IFs that MASM will not assemble).

By default, MASM lists all conditionals.

You may override the .NOCON pseudo-op at assembly time by using the /O function switch in the MASM command line. Refer to Chapter 8 for more information.

Value

You may use .NOCON as a value symbol in your module. The value of .NOCON equals the value of the last **abs-expr** you passed to .NOCON.

The default value for .NOCON is 0.

Example

The source code for this example is:

```
        A=3
        .IFE A  ;FALSE.  MASM LISTS FALSE
        10      ;CONDITIONALS, BY DEFAULT
        20
        30
        .ENDC

        .NOCON 1        ;INHIBIT LISTING OF CONDITIONALS

        .IFE A  ;FALSE. MASM WILL NOT LIST
        40      ;THIS PORTION OF CODE.
        50
        60
        .ENDC

        .IFN A  ;TRUE. MASM LISTS THE ASSEMBLED
        70      ;CODE REGARDLESS OF THE .NOCON
        100     ;SETTING.
        110
        .ENDC
```

The assembly listing for this portion of code is:

```
08         000003       A=3
09         000000       .IFE A  ;FALSE.  MASM LISTS FALSE
10                      10      ;CONDITIONALS, BY DEFAULT
11                      20
12                      30
13                      .ENDC
14
15         000001       .NOCON 1        ;INHIBIT LISTING OF CONDITIONALS
16
17
18         000001       .IFN A  ;TRUE. MASM LISTS THE ASSEMBLED
19 00000'000070         70      ;CODE REGARDLESS OF THE .NOCON
20 00001'000100         100     ;SETTING.
21 00002'000110         110
22                      .ENDC
```

References

"Absolute Expressions" - Chapter 3
"Assembly Listing" - Chapter 4
"Command Line Switches" (/O) - Chapter 8
"Repetitive and Conditional Assembly" - Chapter 6

.NOLOC

Inhibit or re-enable the listing of source lines without location fields.

Syntax

.NOLOC□abs-expr

Purpose

The .NOLOC pseudo-op directs the assembler to either inhibit or enable the listing of source lines that lack a location field. That is, .NOLOC controls the listing of source lines that would not have a location listed in the output. If **abs-expr** evaluates to a nonzero value, the assembler inhibits listing; if **abs-expr** equals zero, listing occurs. **Abs-expr** must be an absolute expression.

By default, the assembler lists all source lines, whether they have location fields or not.

You may override the .NOLOC pseudo-op at assembly time by using the /O function switch in the MASM command line. Refer to Chapter 8 for more information.

Value

You may use .NOLOC as a value symbol, in which case it has the value of the last **abs-expr** you passed to .NOLOC.

The default value for .NOLOC is 0.

Example

```
Consider the following source code:

        .TITLE DF       ;MASM LISTS ALL SOURCE
        .NREL  1        ;LINES BY DEFAULT.
        .TXT "ABCDEF"

        .NOLOC 1        ;INHIBIT LISTING OF ASSEMBLY LINES
                        ;THAT LACK LOCATION FIELDS.

        .TXT "GHIJKL"   ;MASM ONLY LISTS THE FIRST LINE.
        LDA 0,X         ;LISTED
        .LOC .+5        ;NOT LISTED
X:      5               ;LISTED
        .END            ;NOT LISTED

The assembly listing for this code is:

                        .TITLE DF       ;MASM LISTS ALL SOURCE
06       000001         .NREL  1        ;LINES BY DEFAULT.
07 00000!040502         .TXT "ABCDEF"
08       041504
09       042506
10       000000
11
12 00004!043510         .TXT "GHIJKL"   ;MASM ONLY LISTS THE FIRST LINE.
13 00010!020406         LDA 0,X         ;LISTED
14 00016!000005 X:      5               ;LISTED
```

References

"Absolute Expressions" - Chapter 3
"Assembly Listing" - Chapter 4
"Command Line Switches" (/O) - Chapter 8

.NOMAC

Inhibit or re-enable the listing of macro expansions.

Syntax

.NOMAC□abs-expr

Purpose

The .NOMAC pseudo-op either inhibits or permits the listing of macro expansions. If **abs-expr** does not equal zero, the assembler does not include macro expansions in the listing; if **abs-expr** equals zero, listing occurs. **Abs-expr** must be an absolute expression.

By default, the assembler includes all macro expansions in the listing output.

You may override the .NOMAC pseudo-op at assembly time by using the /O function switch in the MASM command line. Refer to Chapter 8 for more information.

Value

You may use .NOMAC as a value symbol, in which case it has the value of the last **abs-expr** you passed to .NOMAC.

The default value for .NOMAC is 0.

Examples

```
08                     .MACRO MAC     ;DEFINE MACRO 'MAC'
09                     5
10                     6
11                     %
12
13                     MAC            ;CALL MACRO 'MAC.' BY DEFAULT,
14 00000'000005        5
15 00001'000006        6
16                                    ;MASM LISTS THE EXPANSION.
17
18       000001        .NOMAC 1       ;INHIBIT MACRO EXPANSIONS.
19
20                     MAC            ;MASM DOES NOT LIST THE EXPANSION.
21
22 00004'000100        100            ;ASSEMBLE THE VALUE 100.
```

Our second example shows how to use .NOMAC inside a macro definition string.

```
08                     .MACRO INSIDE    ;DEFINE MACRO 'INSIDE.'
09                     2
10                     3
11                     .NOMAC 1         ;DURING EXPANSION OF MACRO
12                     4                ;'INSIDE,' MASM DOES NOT LIST
13                     5                ;DATA STATEMENTS 4 AND 5.
14                     .NOMAC 0         ;RE-ENABLE LISTING OF MACRO
15                     6                ;EXPANSIONS.
16                     7
17                     %
18
19                     INSIDE           ;CALL TO MACRO 'INSIDE'
20 00000'000002        2
21 00001'000003        3
22       000000        .NOMAC 0         ;RE-ENABLE LISTING OF MACRO
23 00004'000006        6                ;EXPANSIONS.
24 00005'000007        7
```

References

"Absolute Expressions" - Chapter 3
"Assembly Listing" - Chapter 4
"Command Line Switches" (/O) - Chapter 8
"Listing of Macro Expansions" - Chapter 5

.NREL

Specify normal relocation.

Syntax

.NREL<□abs-expr>

Purpose

The .NREL pseudo-op directs the Macroassembler to locate the following code in normal relocatable (NREL) memory. NREL memory begins immediately after lower page zero (ZREL) and extends from location 400_8 to 32K-1. Address 32K-1 is the upper limit of user-available memory under AOS.

Using .NREL, you may specify one of two predefined memory partitions. These two partitions provide either shared or unshared code and data.

Use a shared partition if you want more than one process to share a single copy of your program at the same time. Unshared partitions are accessible to your process only.

If abs-expr is not present, the location counter is set to unshared code relocation. If the optional expression is present, it is evaluated. If the result is zero, the location counter is set to unshared code relocation; if non-zero, the location counter is set to shared code relocation.

If you leave an NREL partition and later return, the Macroassembler continues assigning addresses from the point where it left off (see the example).

At link time, all source code in a single NREL partition is contiguous in memory. That is, all code with addresses relative to the same zero base is contiguous.

Link may place NREL partitions anywhere in memory above location 377_8 (hence the term 'relocatable'). Thus, you may use MR instructions that provide 15-bit displacement fields to reference any location in NREL memory. If you use shorter MR instructions, be sure that the referenced address can be represented in the MRI displacement field.

Refer to "Partitions" and "Relocatability" in Chapter 3 for more information.

Example

```
08        000000          .NREL 0 ;UNSHARED CODE (NOTE
09 00000'000001           1       ;THE LOCATION COUNTER AND THE
10 00001'000002           2       ;RELOCATION CODE IN COLUMN 9)
11
12        000001          .NREL 1 ;SHARED CODE
13 00000!000003           3
14 00001!000004           4
15
16        000000          .NREL 0 ;UNSHARED CODE.
17 00002'000005           5       ;MASM CONTINUES ASSIGNING ADDRESSES
18 00003'000006           6       ;WHERE IT LEFT OFF EARLIER
19
20        000001          .NREL 1 ;SHARED CODE
21 00002!000007           7
22 00003!000010           10
```

References

"Absolute Expressions" - Chapter 3
AOS Link User's Manual or
AOS/VS Link and Library File Editor User's Manual
"NREL Partitions" - Chapter 3
"Partition Attributes" - Chapter 3
"Relocatability" - Chapter 3

.OB

Name an object file.

Syntax

.OB□filename

Purpose

The .OB pseudo-op directs the assembler to name the object file **filename.** The assembler appends the object file extension .OB onto **filename** unless that name already ends in .OB.

If more than one .OB pseudo-op appears in the source, the assembler returns an error for those source lines. It names the object file after the first source module on the MASM command line (less the .SR extension, if any, and with the new extension .OB).

If you include the /N switch in the MASM command line, directing the assembler not to produce an object file, the assembler ignores the .OB pseudo-op.

If you specify the /B= switch in the MASM command line, the assembler overrides the .OB pseudo-op, and the object file receives the name following the /B= switch.

In sum, the assembler uses the following hierarchy to name object files:

Priority	Object Filename	Description
1 (highest)	**/B=filename**	A switch on the MASM command line
2	**.OB□filename**	A pseudo-op in the source module
3 (lowest)	**Default name**	The name of the first source module on the MASM command line

One of the primary uses of the .OB pseudo-op is in conditional code assembly. You may direct the assembler to assign a name to the object file according to the evaluation of some expression (see the example).

Example

```
        .IFE A              ;IF THE VALUE OF 'A' EQUALS 0, MASM
        .OB SYS1            ;NAMES THE OBJECT FILE 'SYS1.OB.'
        .ENDC
        .IFN A              ;IF THE VALUE OF 'A' DOES NOT EQUAL
        .OB SYS2            ;Z ERO, MASM NAMES THE OBJECT FILE 'SYS2.OB.
```

References

"Command Line Switches" (/N and /B=) - Chapter 8
"Object File" - Chapter 4

.PASS

Number of assembly pass.

Value

The Macroassembler scans your source code twice during the assembly process. Each scan is called a pass.

The .PASS pseudo-op is a value symbol that returns the current assembly pass number. .PASS equals 0 on assembly pass one and 1 on pass two.

Example

The following example defines parameters A, B, and C for later use in the assembly. Because the values of A, B, and C remain constant, MASM need not assemble them on pass two.

```
        .IFE .PASS
        A=0             ;MASM ASSEMBLES THESE
        B=1             ;STATEMENTS ON PASS
        C=2             ;ONE ONLY.
        .ENDC
```

References

"Command Line Switches" /S - Chapter 8
"Macroassembler Symbol Tables" - Chapter 8

.PENT

Define a procedure entry.

Syntax

.PENT□user-symbol$_1$ . . . <user-symbol$_n$>

Purpose

This pseudo-op defines addresses within general procedures that gain control upon one of the following system calls:

?RCALL, ?KCALL, ?RCHAIN

User-symbol must be defined as a user symbol within the general procedure in which it is declared. It must be unique from entries defined in other procedures.

Example

```
              +------------------------+
              |              SAVE      |
              |               .        |
              |               .        |
              |               .        |
Procedure A   |              ?RCALL    |
              |               .        |
              |               .        |
              |               .        |
              |              RTN       |
              +------------------------+
              +------------------------+
              |  .PENT       B         |
Procedure B   |  B:          SAVE      |
              |               .        |
              |               .        |
              |               .        |
              |              RTN       |
              +------------------------+
```

Note that procedure A does not name B in an .EXTN statement. The ?RCALL system macro generates the necessary external references.

.PENT (continued)

Reference

"Intermodule Communication" - Chapter 6

.POP

Pop the value and relocation property of the last expression pushed onto the stack.

Syntax

.POP

Purpose

The .POP pseudo-op removes the the top entry from the Macroassembler stack.

Value

The permanent symbol .POP has the value and relocation property of the last expression you placed on the assembler stack (using .PUSH). When you use .POP, the assembler removes the top entry from the stack.

If the assembler stack contains no values, then .POP has the value 0 and the absolute relocation property. In addition, MASM returns a stack error (O) for that source statement.

Example

```
08       000025        A=25     ;DEFINE 'A'
09 00000'000025        A        ;ASSEMBLE THE PRESENT VALUE OF 'A'
10
11       000025        .PUSH A  ;PUSH THE VALUE OF 'A' ONTO THE
12       000015        A=15     ;ASSEMBLER'S STACK.  ASSIGN 'A' A
13 00001'000015        A        ;NEW VALUE AND ASSEMBLE THAT VALUE.
14
15       000025        A=.POP   ;ASSIGN 'A' THE VALUE ON TOP OF THE
16 00002'000025        A        ;STACK (RESTORE THE ORIGINAL VALUE)
```

See the .PUSH description for a related example.

References

"Relocatability" - Chapter 3
"Stack Control" - Chapter 6

.PTARG

Generate a procedure descriptor.

Syntax

.PTARG□external-symbol

Purpose

This pseudo-op accepts an **external-symbol** defined in another assembly module as a procedure entry symbol. .PTARG transforms the **external-symbol** into a procedure descriptor that can be passed on the user stack as an argument to one of the general resource calls: ?RCALL, ?KCALL, and ?RCHAIN. If the procedure descriptor is not passed on the top of the stack, then the procedure entry must be given as an in-line argument to the resource call (e.g., ?RCALL procedure-entry-point).

Example

```
        .EXTN NAME
.NAME   .PTARG NAME
        .
        .
        .
        LDA 0,.NAME

        PSH 0,0

        ?RCALL
        .
        .
        .
```

If "NAME" had not been passed on the stack, then the resource call would have been expressed as "?RCALL NAME." In this case, "NAME" would not have been given as the argument to an .EXTN pseudo-op; each resource call system macro generates an appropriate external reference when the procedure entry point is given as an in-line argument.

Reference

"Intermodule Communication" - Chapter 6

.PUSH

Push a value and its relocation property onto the stack.

Syntax

.PUSH□expr

Purpose

The .PUSH pseudo-op allows you to save on the assembler stack the value and relocation property of any valid assembler expression. You may continue to push expressions onto the stack until the stack space is exhausted.

The assembler stack is the push-down type. That is, the last expression you place on the stack is the first one to be removed. Use the .POP pseudo-op to access information on the stack.

Example

```
08        000010     .RDX 8         ;INPUT RADIX IS 8 (I.E.,
09 00000'000010      (.RDX)         ;OCTAL).  ASSEMBLE THE CURRENT
10                                  ;INPUT RADIX VALUE.
11        000010     .PUSH .RDX     ;SAVE THE INPUT RADIX ON THE STACK.
12        000012     .RDX 10        ;SET THE INPUT RADIX TO 10 (DECIMAL).
13 00001'000012      (.RDX)         ;ASSEMBLE THE CURRENT INPUT
14                                  ;RADIX VALUE.
15        000010     .RDX .POP      ;SET THE INPUT RADIX TO THE VALUE ON
16                                  ;TOP OF THE STACK (IN THIS CASE 8).
17 00002'000010      (.RDX)         ;ASSEMBLE THE CURRENT INPUT
18                                  ;RADIX VALUE.
```

See the .POP description for a related example.

References

"Expressions" - Chapter 2
"Relocatability" - Chapter 3
"Stack Control" - Chapter 6

.RDX

Set radix for numeric input conversion.

Syntax

.RDX□abs-expr

Purpose

The .RDX pseudo-op defines the radix (base) that MASM uses to interpret the numeric expressions in your source module. For example, if you specify an input radix of 16_{10}, the assembler interprets all numeric expressions in your module in hexadecimal.

The assembler always interprets **abs-expr** in decimal. This argument must be an absolute expression and its value must be in the following range:

2_{10}□<=□abs-expr□<=□20_{10}

If you do not specify an input radix in your module, the default value is 8_{10} (i.e., octal).

If you specify a radix greater than 10, you must use letters to represent values greater than 10 but less than the specified radix. For example, if you declare an input radix of 16_{10} (hexadecimal), use the digits 0 through 9 to represent the quantities 0 through 9_{10}, and use the letters A through F to represent the values 10 through 15_{10}. Table 7-3 shows the correspondence between numeric representations in various bases.

If the input radix is greater than 10, your numeric expressions might start with letters. In these cases, you must place a 0 before the initial letter of the numeric expression to distinguish it from a symbol. For example, if you specify an input radix of 16_{10} (.RDX 16), then you should express the value for decimal 15 as 0F, not simply F.

Regardless of the input radix, the assembler interprets any number containing a decimal point as decimal. For example, the numeric expression 12. always equals 12_{10}, regardless of the input radix. This feature allows you to combine decimal numbers in expressions with numbers of other radixes (e.g., 0F+12.-3A2).

Note that the input and output radixes are entirely distinct. Setting the input radix does not affect the output radix (set with .RDXO).

Table 7-3. Numeric Representations in Various Bases

Octal (Base 8)	Decimal (Base 10)	Hexadecimal (Base 16)
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
10	8	8
11	9	9
12	10	A
13	11	B
14	12	C
15	13	D
16	14	E
17	15	F
20	16	10

Value

You may use .RDX as a value symbol. In this case, .RDX has the value of the current input radix.

Example

In the following example, the output radix is 8 (i.e., octal). You may alter this setting with .RDXO pseudo-op.

```
08        000010         .RDX 8  ;INPUT RADIX IS 8.
09 00000'000123          123     ;MASM INTERPRETS 123 AS OCTAL.
10
11        000012         .RDX 10 ;SET INPUT RADIX TO 10.
12 00001'000173          123     ;MASM INTERPRETS 123 AS DECIMAL.
13
14        000020         .RDX 16 ;SET INPUT RADIX TO 16
15 00002'000443          123     ;MASM INTERPRETS 123 AS HEXADECIMAL.
16 00003'000017          0F      ;NOTE THE LEADING ZERO.
17 00004'000173          123.    ;MASM INTERPRETS 123 AS DECIMAL EVEN
18                               ;THOUGH THE INPUT RADIX IS 16.
19
20 00005'000020          (.RDX)  ;ASSEMBLE THE CURRENT INPUT
21                               ;RADIX VALUE.
```

References

"Absolute Expressions" - Chapter 3
"Numbers" - Chapter 2
"Radix Control" - Chapter 6

.RDXO

Set the radix for numeric output conversion.

Syntax

.RDXO□abs-expr

Purpose

The .RDXO pseudo-op defines the radix (base) that the assembler uses to represent numeric expressions in the assembly listing. For example, if you specify an output radix of 10_{10}, the assembler presents all locations and values in decimal, regardless of the input radix.

The assembler always interprets **abs-expr** as decimal. This argument must be an absolute expression, and its value must be in the following range:

$$8_{10} <= \textbf{abs-expr} <= 20_{10}$$

If you do not specify an output radix in your module, the assembler uses 8_{10} (i.e., octal).

Table 7-3 in the .RDX pseudo-op description shows the correspondence between numeric representations in various bases.

Value

You may use .RDXO as a value symbol, in which case it equals the current output radix. The assembler always expresses the current output radix as '10'.

Example

```
08        000010      .RDX 8        ;INPUT RADIX IS 8.
09        000010      .RDXO 8       ;OUTPUT RADIX IS 8.
10 00000'000077       77
11 00001'000022       22
12 00002'000045       45
13
14        000012      .RDX 10       ;INPUT RADIX IS 10.
15         00010      .RDXO 10      ;OUTPUT RADIX IS 10.
16 00003' 00077       77
17 00004' 00022       22
18 00005' 00045       45
19
20         00016      .RDX 16       ;INPUT RADIX IS 16.
21          0010      .RDXO 16      ;OUTPUT RADIX IS 16.
22 00006'  0077       77
23 00007'  0022       22
24 00008'  0045       45
25
26                                  ;INPUT RADIX IS 16.
27        000010      .RDXO 8       ;OUTPUT RADIX IS 8.
28 00011'000167       77
29 00012'000042       22
30 00013'000105       45
31
32                                  ;INPUT RADIX IS 16.
33         00010      .RDXO 10      ;OUTPUT RADIX IS 10.
34 00012' 00119       77
35 00013' 00034       22
36 00014' 00069       45
37 00015' 00010       (.RDXO)       ;ASSEMBLE THE CURRENT VALUE
38                                  ;OF THE OUTPUT RADIX.  MASM
39                                  ;ALWAYS REPRESENTS THIS AS 10,
40                                  ;REGARDLESS OF THE CURRENT BASE.
```

References

"Absolute Expressions" - Chapter 3
"Assembly Listing" - Chapter 4
"Numbers" - Chapter 2
"Radix Control" - Chapter 7

.REV

Set revision level.

Syntax

.REV□major-revision-number□minor-revision-number

Purpose

The .REV pseudo-op specifies your program's revision level. Generally, you use this pseudo-op when you want to keep track of different versions of the same program.

Arguments must be absolute expressions with values in the range 0-255_{10}. The assembler uses the current input radix to evaluate these expressions. When issuing .REV, you must supply values for both arguments.

The revision level you indicate in your source module passes into the object file and then into the program file. If MASM encounters more than one .REV pseudo-op during an assembly, the object file receives the level specified in the last .REV statement.

Use the CLI REVISION command to obtain the revision level of any executable program file.

Example

The source code (in the file MNTS) for this example is:

```
        .TITLE MNTS
        .REV 12,5     ;MASM INTERPRETS THE REVISION LEVEL IN THE
                      ;CURRENT INPUT RADIX (OCTAL BY DEFAULT).
        .NREL
```

When properly bound and linked, you can obtain the revision number from the CLI as follows:

```
)REVISION MNTS.PR<nl>
10.05
```

References

"Absolute Expressions" - Chapter 3
AOS Command Line Interpreter User's Manual- REV command
AOS Link User's Manual - /REV switch or
AOS/VS Link and Library File Editor User's Manual

.TITL

Entitle an object module.

Syntax

.TITL□user-symbol

Purpose

The .TITL pseudo-op provides a name for your object module by placing **user-symbol** in the module's title block (described in the AOS Link User's Manual or AOS/VS Link and Library File Editor User's Manual).

The title you assign in the module appears at the top of each page in the assembly listing. **User-symbol** need not be unique from other symbols in your source module.

If you do not include .TITL in your source module, the assembler supplies the name ".MAIN" as default.

Note that the name you assign in the .TITL statement has no relation to the name of the object file (see the .OB pseudo-op).

Example

```
.TITLE MOD1
   •
   •
   •
```

References

"Assembly Listing" - Chapter 4
AOS Link User's Manual or
AOS/VS Link and Library File Editor User's Manual
"User Symbols" - Chapter 2

.TOP

Value and relocation of last stack expression.

Value

.TOP has the value and relocation property of the last expression pushed to the variable stack. .TOP differs from .POP in that it does not remove (pop) the last pushed expression from the stack. If no expressions are pushed, zero (absolute relocation) is returned and the overflow flag (O) is given.

Example

```
08        000010          .PUSH 10        ;PLACE 10 ON STACK
09        000020          .PUSH 20        ;PLACE 20 ON STACK
10        000030          .PUSH 30        ;PLACE 30 ON STACK
11
12               ;STACK IS NOW 30 20 10
13
14 00000'000030           .TOP            ;TOP OF STACK
15 00001'000030           .TOP            ;TOP OF STACK (UNCHANGED BY
16                                        ;PREVIOUS '.TOP')
17 00002'000030           .POP            ;STACK POP
18
19               ;STACK IS NOW 20 10
20
21 00003'000020           .TOP            ;NEW STACK TOP
```

Reference

"Stack Control" - Chapter 6

.TSK

Reserve a number of tasks.

Syntax

.TSK□abs-expr

Purpose

The .TSK pseudo-op specifies the maximum number of tasks that your program can initiate at execution time. The argument you pass to .TSK must be an absolute expression and must be less than or equal to 32_{10}.

If several object files in the same Link command line contain .TSK declarations, Link uses the largest.

You may override the .TSK pseudo-op at link time by using the /TASKS= switch in the Link command line.

Example

```
.TITLE MOD       ;THIS PROGRAM MAY INITIATE
.TSK 5           ;UP TO 5 TASKS AT RUNTIME.
```

References

"Absolute Expressions" - Chapter 3
Advanced Operating System (AOS) User's Manual
AOS Programmer's Manual

.TXT, .TXTE, .TXTF, .TXTO

Store and specify a text string.

Syntax

```
.TXT□*string*
.TXTE□*string*
.TXTF□*string*
.TXTO□*string*
```

Purpose

These pseudo-ops direct the assembler to store the octal equivalent of an ASCII text string in consecutive memory words. In the above syntax descriptions, **string** is an ASCII text string and * represents a character that you use to delimit the string. The delimiter may be any character <u>except</u>

- a character that appears in text string **string**, or
- carriage return, form feed, NEW LINE, tab, space, null, or rubout.

You may use carriage return, form feed, and NEW LINE to continue a string from line to line, but the assembler will not store these characters as part of the text string.

The assembler interprets the first character after the break (□) as **string's** delimiter. The assembler then stores the following characters in pairs in consecutive memory words until it encounters the delimiting character again. That is, the assembler generates one 16-bit storage word for every two characters in **string.**

Storage of a character of a string requires seven bits of an eight-bit byte. The leftmost (parity) bit may be set to 0, 1, even parity, or odd parity as follows:

.TXT	Sets leftmost bit to 0 unconditionally.
.TXTF	Sets leftmost bit to 1 unconditionally.
.TXTE	Sets leftmost bit for even parity on byte.
.TXTO	Sets leftmost bit for odd parity on byte.

The assembler allocates an 8-bit byte for each character (i.e., two characters per 16-bit word). By default, the assembler packs the characters of your string from left to right within memory storage words. You may alter this packing mode with the .TXTM pseudo-op.

If your string contains an odd number of characters, the assembler pairs a null (all zero) byte with the last character of the string. If your string contains an even number of characters, the assembler stores a null word (2 null bytes) immediately after the string. You may suppress this null storage word by using the .TXTN pseudo-op.

Within the string, angle brackets(< >) can be used to delimit an arithmetic expression. The expression will be evaluated, masked to seven bits, and the eighth bit set as specified by the pseudo-op. Note that _no_ logical operators are permitted within the expression.

By using angle brackets, you may store the ASCII codes for characters that you could not otherwise include in your text string. For example, you may not include a NEW LINE character in your text string. However, you may include the ASCII code for NEW LINE in a text string if you enclose that value in brackets.

```
     .TXT      "A<12>"
```

This statement directs the assembler to generate a storage word that contains the ASCII codes for "A" (101_8) and "NEW LINE" (12_8). By default, the Macroassembler stores a null (all zero) word in the following location.

.TXT, .TXTE, .TXTF, .TXTO (continued)

Example

```
08       000000           .TXTM 0        ;SET BYTE PACKING RIGHT/LEFT.
09 00000'041101           .TXT #ABC D#   ;EACH EXAMPLES STORES ASCII CODE
10       020103
11       000104
12 00003'041101           .TXTE /ABC D/  ;FOR THE TEXT STRING "ABC D" IN MEMORY.
13       120303
14       000104
15 00006'141301           .TXTF @ABC D@  ;NOTE THE VARIOUS CHARACTERS WE
16       120303
17       000304
18 00011'141301           .TXTO EABC DE  ;USE FOR TEXT DELIMITERS.
19       020103
20       000304

See .TXTM and .TXTN for related examples.
```

References

"Absolute Expressions" - Chapter 3
"ASCII Character Set" - Appendix A
"Special Integer-Generating Formats" - Chapter 2
"Text Strings" - Chapter 6

.TXTM

Change text byte packing.

Syntax

.TXTM□abs-expr

Purpose

The .TXTM pseudo-op directs the assembler to pack bytes either left to right or right to left within memory words when it encounters a .TXT, .TXTE, .TXTF, or .TXTO pseudo-op. If **abs-expr** evaluates to zero, the assembler packs the bytes right to left; if **abs-expr** does not equal zero, the assembler packs bytes left to right. The argument you supply to .TXTM must be an absolute expression.

If you do not use the .TXTM pseudo-op in your module, MASM packs bytes from left to right, by default.

Value

You may use .TXTM as a value symbol. In this case, .TXTM represents the value of the last **abs-expr** you supplied to it.

The default value for .TXTM is 1.

Example

```
07          0010     .RDXO 16        ;OUTPUT RADIX HEXADECIMAL
08 00000'   4142     .TXT "ABCDE"    ;PACK BYTES LEFT/RIGHT
09          4344
10          4500
11                                   ;WITHIN WORDS, BY DEFAULT.
12 00003'   0001     (.TXTM)         ;ASSEMBLE THE CURRENT VALUE OF
13                                   ;.TXTM (1, BY DEFAULT).
14          0000     .TXTM 0         ;PACK BYTES RIGHT/LEFT.
15 00004'   4241     .TXT "ABCDE"
16          4443
17          0045
18 00007'   0000     (.TXTM)         ;ASSEMBLE THE CURRENT VALUE
19                                   ;OF .TXTM.
```

.TXTM (continued)

References

"Absolute Expressions" - Chapter 3
"ASCII Character Set" - Appendix A
"Text Strings" - Chapter 6

.TXTN

Determine text string termination.

Syntax

.TXTN□abs-expr

Purpose

The .TXTN pseudo-op specifies whether or not the assembler will place a null word at the end of a .TXT, .TXTE, .TXTF, or .TXTO text string that contains an even number of characters.

If **abs-expr** evaluates to zero, all text strings containing an even number of characters terminate with a 16-bit null word (all zeros). If **abs-expr** does not equal zero, the assembler does not place a null word after the last two characters in your string. The argument you supply to .TXTN must be an absolute expression.

If you do not use .TXTN in your module, the assembler terminates even length text strings with a null word, by default. When a string contains an odd number of characters, the assembler stores a null byte with the last character, in all cases.

Value

You may use .TXTN as a value symbol. In this case, .TXTN represents the value of the last **abs-expr** you passed to it.

The default value for .TXTN is 0.

.TXTN (continued)

Example

```
08 00000'030462      .TXT "1234"    ;TERMINATE EVEN-LENGTH STRINGS
09        031464
10        000000
11                                  ;WITH A NULL WORD, BY DEFAULT.
12 00003'000000      (.TXTN)        ;ASSEMBLE THE CURRENT VALUE OF
13                                  ;.TXTN (0, BY DEFAULT).
14        000001     .TXTN 1        ;DO NOT ADD A NULL WORD TO THE
15                                  ;END OF EVEN-LENGTH STRINGS.
16 00004'030462      .TXT "1234"
17        031464
18 00006'000001      (.TXTN)        ;ASSEMBLE THE CURRENT VALUE OF
19                                  ;.TXTN (1).
```

References

"Absolute Expressions" - Chapter 3
"Text Strings" - Chapter 6

.XPNG

Delete symbol and macro definitions.

Syntax

.XPNG

Purpose

This pseudo-op removes all semipermanent symbol definitions (macro and instruction definitions) from the assembler's symbol table. .XPNG is used primarily as follows:

1. You write a program containing .XPNG followed by definitions of any semipermanent symbols.
2. The program is assembled using the function switch /S in the MASM command line. This causes the assembler to stop assembly after pass 1 and save the symbols in MASM.PS (or MASM16.PS).
3. You can then use the modified assembler containing permanent symbols and those semipermanent symbols defined in Step 2.

Example

```
The following source code is contained in file XP.

        .TITLE XP

        .XPNG                   ;REMOVE SEMIPERMANENT SYMBOLS
                                ;FROM SYMBOL TABLE.
        .DALC SUB=102400        ;DEFINE 'SUB' USING .DALC.
        .DMRA LDA=20000         ;DEFINE 'LDA' USING .DMRA.
        .DMRA STA=40000         ;DEFINE 'STA' USING .DMRA.

        .END
```

.XPNG (continued)

This code is assembled with the following CLI command:

```
        { MASM/S   }
    )X  {          }  XP
        { MASM16/S }
```

After assembly, the assembler's symbol table contains definitions for SUB, LDA, and STA, but for no other semipermanent symbols.

References

"Macroassembler Symbol Tables" - Chapter 8
"Symbol Tables" - Chapter 3
"Symbols" - Chapter 2

.ZREL

Specify lower page zero relocation.

Syntax

.ZREL

Purpose

The .ZREL pseudo-op directs the assembler to assign relocatable addresses in lower page zero to subsequent source lines in your module; i.e., to assign locations in the predefined ZREL memory partition. Lower page zero relocatable (ZREL) memory extends from location 50_8 to 377_8. Thus, you may express any ZREL location in a displacement field of 8 or more bits (in any MR instruction).

The words following the .ZREL pseudo-op receive relocatable addresses starting with zero. If you leave the ZREL partition during an assembly and later return, the assembler continues assigning ZREL addresses at the point where it left off.

At link time, all code in the ZREL memory partition is contiguous in memory.

Example

```
                         .TITLE Z
06                       .ZREL    ;PLACE THE FOLLOWING CODE IN LOWER
07 00000-000100          100      ;PAGE ZERO RELOCATABLE (ZREL) MEMORY
08 00001-000200          200      ;(NOTE THE ADDRESS RELOCATION FLAG
09 00002-000300          300      ;IN COLUMN 9).
10       000000          .NREL 0  ;UNSHARED NORMAL RELOCATABLE (NREL)
11 00000'000400          400      ;MEMORY
12 00001'000500          500
13                       .ZREL    ;LOWER PAGE ZERO RELOCATABLE (ZREL)
14 00003-000600          600      ;MEMORY. NOTE THAT MASM CONTINUES
15 00004-000700          700      ;ASSIGNING ZREL ADDRESSES AT THE
16                                ;POINT WHERE IT LEFT OFF EARLIER.
```

References

AOS Link User's Manual or
AOS/VS Link and Library File Editor User's Manual
"Relocatability" - Chapter 3
"ZREL Partition" - Chapter 3

End of Chapter

Chapter 8
Macroassembler Operating Procedures

Masm Command Line

The CLI command line that invokes the Macroassembler is

```
     ( MASM   )
XEQ  {        } <function-switch...>□pathname<arg-switch>
     ( MASM16 )
```

where:

XEQ is a CLI command that executes a program. The single character X is an acceptable abbreviation for XEQ.

MASM is the name of the Macroassembler program (without the .PR extension). Use **MASM16** if you are assembling 16-bit programs under AOS/VS.

function-switch is one or more of the optional function switches (see Table 8-1).

pathname is the pathname of a source file. You must include at least one source file in each **MASM** command line. If you include more than one source file, make sure that all but the last one end with the .EOF pseudo-op; the last file should end with .END.

arg-switch is the optional argument switch /S (see Table 8-2).

When you issue the MASM command, the Macroassembler assembles one or more source files (**pathnames**) and produces an object file and a variety of listings (depending on which function switches you use). The object file is not executable; you must use the Link utility to produce a program file (see "Linking and Executing Your Program" in this chapter.)

By default (i.e., if you do not use any function switches), the object file bears the name of the first filename in the command line (see "Filenames"). Also by default, the Macroassembler does not produce an assembly listing; it reports all assembly errors to the generic file **@OUTPUT.** We describe the object file, assembly listing, and other forms of Macroassembler output in Chapter 4.

An incorrect MASM command line generates a command line error. Refer to Appendix C for descriptions of all command line errors.

Command Line Switches

You may use two types of switches in the MASM command line:

* Function switches

* Argument switches

A function switch appears after the word MASM and provides information global to the current assembly. An argument switch appears after the name of a source file and provides information local to that particular file.

The syntax in the previous section provides information about the placement of switches in the MASM command line. Note that all switches appear immediately after the term they modify; do not insert any spaces before a switch. There is no limit to the number of switches that may appear in a single command line.

Table 8-1 describes the function switches you may use in the MASM command line. For a complete discussion of the various forms of Macroassembler output, refer to Chapter 4. For a description of permanent, instruction, macro, and user symbols, see "Symbols" in Chapter 2.

Table 8-1. AOS MASM Function Switches

Function Switch	Action
/8	This switch tells the Macroassembler to examine the first eight characters when it resolves symbols instead of the first five characters. Note that macro names must still be unique in their first five characters when this switch is used. Also, any symbol that is not a macro name must not have its first five characters the same as those of a macro name. The Macroassembler will not give a warning or error if this restriction is violated.
/B=filename	Name the object file filename.OB instead of the name of the first source file in the assembly command line. If the specified filename already has the .OB extension, the Macroassembler does not add a second .OB extension.
/E	Do not write pass 2 error messages to @OUTPUT, unless there is no listing file. Some pass 1 error messages are automatically written to @OUTPUT. Error codes are described in Appendix C.
/E=filename	Creates file filename and reports all assembly errors to that file.
/F	Generate or suppress a form feed as necessary to produce an even number of listing pages. By default, a form feed is generated after each listing page.
/HASH=	Directs the Macroassembler to set the size of its hash table to the value specified in the switch. The hash table's default size is 32 entries. The switch accepts values of 2 to 128 in powers of two (e.g., 2, 4, 8, 16, 32, 64, or 128).
/K	Keep the Macroassembler's temporary symbol file table (MASM.ST.TMP) after the assembly is complete. By default, the symbol table is deleted.

(continues)

Table 8-1. AOS MASM Function Switches

Function Switch	Action
/L	Write a listing to the current list file. The new listing will follow any listings that are already in the file. Listings always include a cross reference of symbols in the program. The cross reference shows the page and line number where each symbol is used. If you use the /L switch, program MASMXR.PR must be present in the same directory as the Macroassembler itself.
/L=filename	Write a listing file to the file specified by filename.
/M	Flag redefinition of permanent symbols as multiple-definition (M) errors.
/MEM=	Directs the Macroassembler to allocate the number of buffers specified in the switch. The Macroassembler uses 12 buffers by default. The switch accepts values of 1 to 32.
/N	Do not produce an object file. The Macroassembler performs all operations (i.e., checks for errors, produces listings, etc.) but does not produce a binary object-code file. You usually use this switch to locate errors in your source code.
/O	Override all listing suppression controls (.NOCON, .NOLOC, .NOMAC, as well as double asterisks (**)).
/P	Add semipermanent symbols to the cross-reference listing. By default, they are not included.
/PS=filename	Directs the Macroassembler to use filename as the permanent symbol table for the current assembly. The Macroassembler's default symbol table is MASM.PS (or MASM16.PS if you are using MASM16).
/R	Produce a binary object (.OB) file even if there are assembly errors in the source files. By default, the Macroassembler produces no .OB file when there are assembly errors.

(continued)

Table 8-1. AOS MASM Function Switches

Function Switch	Action
/S	Tells the Macroassembler to skip pass 2, and save a version of the symbol table and macro definitions in =MASM.PS (or =MASM16.PS). No .OB file is produced.
/U	Include user symbols in the binary object (.OB) file. This lets the Debugger locate user symbols in your program.
/Z	Prints the DGC proprietary license heading at the top of each assembly and cross-reference page. By default, this heading is not printed. Note that this switch is useful to DGC personnel only.

(concluded)

Table 8-2 describes the /S argument switch (do not confuse this switch's description with the /S function switch described above). The /S argument switch may appear after any source filename in the MASM command For example:

$$\textbf{XEQ}\begin{Bmatrix}\textbf{MASM}\\ \textbf{MASM16}\end{Bmatrix}\ \textbf{MOD1}\quad\textbf{MOD2/S}\quad\textbf{MOD3}\ \langle\textbf{nl}\rangle$$

Table 8-2. MASM Argument Switch

Argument Switch	Action
pathname/S	Tells MASM not to process this source file on assembly pass two. This switch allows you to use a source file to define symbols but not generate object code. /S does not hinder the assembly of other source files in the command line. You typically use this switch with files that define parameters and macros. In these cases, the /S switch reduces assembly time and decreases the size of the assembly listing. "Macroassembler Symbol Tables" in this chapter explains how the Macroassembler resolves symbols.

Linking and Executing Your Program

The Macroassembler output is not executable. You must first process your object file(s) with the Link utility. The general Link command line is

```
    XEQ    LINK   file ... <nl>
```

where:

file is the pathname of an object file. You need not specify the .OB extension

This command generates an executable file named file.PR. If you include more than one object file in the Link command line, the program file receives the name of the first one, by default.

The Link utility offers many options that we do not present in this manual. For example, you may include a variety of switches in the command line, and you may also link library files along with your object files. For a complete description of these and other features of the Link utility, refer to the AOS Link User's Manual or the AOS/VS Link and Library File Editor User's Manual.

To execute a program file generated by Link, issue the command

```
    XEQ    file
```

where:

file is the pathname of a program file; you need not specify

Filenames

Table 8-3 summarizes the AOS file-naming conventions. Note that the table lists only the filename extensions, not the complete filename.

Table 8-3. AOS Filename Extensions

Extension	Contents of File
.OB	Object file (generated by MASM)
.PR	Program (executable) file (generated by Link)
.PS	Permanent symbol table (generated by MASM)
.SR	Assembly language source file

Normally, the names of your source files end with the .SR extension; e.g., filename.SR. However, you need not specify the .SR extension in the MASM command line; e.g., filename is sufficient. The Macroassembler always searches for filename.SR first. If MASM does not find this file, it searches for filename.

For example, the following two command lines are functionally equivalent:

```
    ( MASM   )
XEQ {        }  FILE1   FILE2 <nl>
    ( MASM16 )

    ( MASM   )
XEQ {        }  FILE1.SR   FILE2.SR <nl>
    ( MASM16 )
```

The object file normally receives the name of the first source file in the command line, without the .SR extension (if any) and with the .OB extension. You may specify a different name for the object file by using the /B= function switch or the .OB pseudo-op. Table 8-4 shows the file-naming priority employed by the Macroassembler.

Table 8-4. Object Filename

Priority	Object Filename	Description
1 (highest)	**/B=filename**	The object file receives the name you specify with the /B= function switch on the MASM command line
2	**.OB}filename**	The object file receives the name you specify in an .OB pseudo-op in a source file
3 (lowest)	**Default filename**	The object file receives the name of the first source file on the MASM command line

The following examples will help clarify these naming conventions:

```
    ( MASM   )
XEQ {        }  FILE1  FILE2 <nl>
    ( MASM16 )
```

```
    ( MASM   )
XEQ {        }  FILE1.SR  FILE2.SR <nl>
    ( MASM16 )
```

Both of the above command lines produce an object file with the name FILE1.OB.

```
    ( MASM   )
XEQ {        } /B=BOND  FILE1  FILE2 <nl>
    ( MASM16 )
```

This command generates an object file named BOND.OB. The Macroassembler adds the extension .OB to a specified filename only if the extension is not already present. Thus,

```
    ( MASM   )
XEQ {        }/B=BOND.OB  FILE1  FILE2 <nl>
    ( MASM16 )
```

also produces an object file named BOND.OB.

Generic Filenames

Generic filenames simplify the use of certain common files and devices. By using a generic name (e.g., @LIST, @OUTPUT), you need not specify a particular file or device when developing your program. Rather, you can associate the generic filename with a specific file or device at runtime. In this manner, the same program can access a different file or device each time you execute it.

For further information on generic files, refer to the AOS Programmer's Manual and the AOS and AOS/VS Command Line Interpreter User's Manual.

Macroassembler Symbol Tables

A primary function of the Macroassembler is to translate the symbols in your source program into binary code. The Macroassembler uses its internal database and the temporary symbol table during this process.

The Macroassembler's database defines all permanent symbols. Permanent symbols consist of the Macroassembler's pseudo-op descriptions. These definitions reside inside the Macroassembler, and are present during all assemblies. You cannot change pseudo-op definitions.

The permanent symbol table resides in an external disk file. This file contains definitions for all the symbols you use but do not define in your source (except pseudo-ops). The permanent symbol table also contains definitions for the standard assembly language instructions and AOS system calls and parameters. In most cases, the permanent symbol table resides in disk file MASM.PS (or MASM16.PS if you are using MASM16).

When you issue a MASM command line, the Macroassembler creates a temporary symbol table. Initially, this table contains a copy of the permanent symbol table. At the end of the assembly, the temporary symbol table contains these original definitions, plus the definitions of symbols defined in your source code. The Macroassembler normally deletes this table at the end of an assembly. However, you can save the table by using the /K function switch.

In Chapter 3, we explained how the Macroassembler uses these tables to resolve the symbols in your source module. The following discussion reviews this process and then explains how to create a permanent symbol table.

Symbol Resolution

The following outline describes how the Macroassembler resolves the symbols in your program. Figure 3-1 contains a flowchart of this process.

When the Macroassembler encounters a symbol in your source code, it searches its database and the temporary symbol table for the symbol's definition.

Remember that the Macroassembler database contains definitions for the pseudo-op symbols. The Macroassembler places a copy of the permanent symbol table into the temporary symbol table at the start of the assembly process.

If a symbol is a pseudo-op, MASM does not search the temporary symbol table. After checking the temporary symbol table, the Macroassembler checks to see if the symbol is defined in the current source line. If so, MASM copies that definition into the temporary symbol table. If the temporary table already defines the symbol, MASM updates the symbol's definition.

On pass two, the Macroassembler uses its database and the temporary symbol table to substitute binary code for the symbols in your source. If the Macroassembler cannot find a symbol's definition, it returns an error (unless you used the .EXTU pseudo-op in your source program).

The above outline summarizes a discussion that appears earlier in this manual. For more information, see "Symbol Interpretation" in Chapter 3.

Permanent Symbol Table

The <u>permanent symbol table</u> resides in a disk file. It should contain definitions for all symbols that you use in your program and that are not defined in either your source code.

In most cases, you use the permanent symbol table we provide with the Macroassembler program. This file resides in disk file MASM.PS (or MASM16.PS). The default .PS file is made from four other files: EBID.SR, ECID.SR, SYSID.SR (SYSID.16.SR), and PARU.SR (PARU.16.SR). EBID.SR and ECID.SR contain definitions for the 16-bit ECLIPSE computers' assembly language instructions. The SYSID and PARU files contain operating system call definitions and user parameter definitions. System calls include ?OPEN, ?READ, and ?TASK; system parameters include ?ISTI and ?TLNK. The <u>AOS Programmer's Manual</u> describes these symbols in detail.

The Macroassembler uses file MASM.PS (or MASM16.PS) as the permanent symbol table by default. The following sections of this chapter explain how to create a new permanent symbol table and how to specify a table other than MASM.PS (or MASM16.PS) for an assembly.

Building a Permanent Symbol Table

To create a permanent symbol table, you must use the /S function switch on the assembly command line. This switch directs the assembler to copy all the definitions in your source module(s) into the temporary symbol table during pass one. The assembler then stores the temporary symbol table in disk file MASM.PS (or MASM16.PS). It first deletes the existing .PS file from your current directory, if one is present. Thus, at the end of pass one, the permanent symbol table in MASM.PS (or MASM16.PS) contains definitions for all the symbols you define in your source module(s).

The general MASM command for creating a permanent symbol table is

```
      ( MASM   )
XEQ  <          > /S       sourcefile ...<nl>
      ( MASM16 )
```

where:

sourcefile contains definitions for all the symbols you want in the permanent symbol table. If you include more than one source file in an assembly command line, make sure that all but the last one end with the .EOF pseudo-op; the last file should end with .END.

To create a permanent symbol table that contains definitions for all AOS system calls and system parameters, issue the following command:

```
      ( MASM   )
XEQ  <          > /S  PARU.SR SYSID.SR<nl>
      ( MASM16 )
```

Normally, you need not issue this command line since we provide a copy of this permanent symbol table with the Macroassembler program (in file MASM.PS (or MASM16.PS)).

The Macroassembler places the permanent symbol table in file MASM.PS (or MASM16.PS), by default. If you want to place the permanent table in a different file, use both the /S and /PS= switches in the MASM command line as follows:

```
      ( MASM   )
XEQ  <          >/S/PS=filename  sourcefile ...<nl>
      ( MASM16 )
```

where:

filename is the file that will contain the permanent symbol table

sourcefile contains definitions for all the symbols you want in the permanent symbol table

The above command line directs the Macroassembler to copy all the symbol definitions in **sourcefile** into disk file **filename.** If **filename** exists before the assembly, the Macroassembler deletes that file before creating your permanent symbol table.

Specifying a Permanent Symbol Table for an Assembly

After you build a permanent symbol table, you may use that table during the assembly of your source modules. Specify a particular symbol table file by using the /PS= switch in the MASM command line. The Macroassembler then uses that permanent symbol table to resolve the symbols in your source module. For example:

```
    { MASM   }
XEQ {        }/PS=A.PS  SOURCEFILE <nl>
    { MASM16 }
```

The Macroassembler uses file A.PS as the permanent symbol table when resolving symbols in SOURCEFILE (as described earlier in this chapter).

If you do not use the /PS= switch to specify a permanent symbol table, the Macroassembler uses file MASM.PS (or MASM16.PS), if it is available.

Permanent Symbol Table Size

The permanent symbol table may include approximately $8,000_{10}$ symbols and their definitions. This figure assumes that you do not include any macro definition strings. The more macro text you place in the table, the smaller the amount of space available for symbol definitions.

Increasing symbol length does not decrease the number of symbols you may define. The following section describes symbol length in depth.

Symbol Length

The permanent symbol table specifies how many characters the Macroassembler should use to resolve the symbols in your source program. The default value for symbol length is 5 characters.

During assembly, the Macroassembler ignores all excess characters; they do not generate an error. Thus, the Macroassembler views the following three symbols as identical (if the symbol length is 5 characters):

```
LOCAT1        LOCAT2        LOCAT_START
```

To alter the symbol length for an assembly, include the /8 in the MASM command line when building your permanent symbol table. The /8 switch changes the symbol length that the Macroassembler recognizes from 5 characters to 8 characters. For example:

$$\mathbf{XEQ}\begin{Bmatrix}\mathbf{MASM}\\ \mathbf{MASM16}\end{Bmatrix}\mathbf{/S/PS{=}MASM8.PS/8\quad SOURCE1\quad <nl>}$$

This command directs the Macroassembler to create a permanent symbol table and store it in file MASM8.PS. This table will contain all the symbol definitions in file SOURCE1 and will identify those symbols according to their first 8 characters. Thus, each time you specify MASM8.PS as the permanent symbol table, MASM will resolve symbols according to their first 8 characters. For example:

$$\mathbf{XEQ}\begin{Bmatrix}\mathbf{MASM}\\ \mathbf{MASM16}\end{Bmatrix}\mathbf{/PS{=}MASM8.PS\quad SOURCE2\ <nl>}$$

This command directs the Macroassembler to use file MASM8.PS as the permanent symbol table. The Macroassembler will identify the symbols in source file SOURCE2 according to their first 8 characters, as specified in the permanent symbol table.

Remember that macro names must still be unique in their first five characters, even when the /8 switch is used. Also, symbols other than macro names must not have their first five characters the same as those of a macro name. The Macroassembler will not give a warning or error if this restriction is violated.

End of Chapter

Appendix A
ASCII Character Set

To find the *octal* value of a character, locate the character, and combine the first two digits at the top of the character's column with the third digit in the far left column.

LEGEND:

Character code in decimal
EBCDIC equivalent hexadecimal code
Character

	10_	
0	64 7C	@

OCTAL	00_		01_		02_		03_		04_		05_		06_		07_	
0	0 00	NUL	8 16	BS (BACK-SPACE)	16 10	DLE ↑P	24 18	CAN ↑X	32 40	SPACE	40 4D	(	48 F0	0	56 F8	8
1	1 01	SOH ↑A	9 05	HT (TAB)	17 11	DC1 ↑Q	25 19	EM ↑Y	33 5A	!	41 5D	)	49 F1	1	57 F9	9
2	2 02	STX ↑B	10 15	NL (NEW LINE)	18 12	DC2 ↑R	26 3F	SUB ↑Z	34 7F	" (QUOTE)	42 5C	*	50 F2	2	58 7A	:
3	3 03	ETX ↑C	11 0B	VT (VERT. TAB)	19 13	DC3 ↑S	27 27	ESC (ESCAPE)	35 7B	#	43 4E	+	51 F3	3	59 5E	;
4	4 37	EOT ↑D	12 0C	FF (FORM FEED)	20 3C	DC4 ↑T	28 1C	FS ↑\	36 5B	$	44 6B	, (COMMA)	52 F4	4	60 4C	<
5	5 2D	ENQ ↑E	13 0D	RT (RETURN)	21 3D	NAK ↑U	29 1D	GS ↑]	37 6C	%	45 60	-	53 F5	5	61 7E	=
6	6 2E	ACK ↑F	14 0E	SO ↑N	22 32	SYN ↑V	30 1E	RS ↑↑	38 50	&	46 4B	. (PERIOD)	54 F6	6	62 6E	>
7	7 2F	BEL ↑G	15 0F	SI ↑O	23 26	ETB ↑W	31 1F	US ↑←	39 7D	' (APOS)	47 61	/	55 F7	7	63 6F	?

OCTAL	10_		11_		12_		13_		14_		15_		16_		17_	
0	64 7C	@	72 C8	H	80 D7	P	88 E7	X	96 79	\` (GRAVE)	104 88	h	112 97	p	120 A7	x
1	65 C1	A	73 C9	I	81 D8	Q	89 E8	Y	97 81	a	105 89	i	113 98	q	121 A8	y
2	66 C2	B	74 D1	J	82 D9	R	90 E9	Z	98 82	b	106 91	j	114 99	r	122 A9	z
3	67 C3	C	75 D2	K	83 E2	S	91 8D	[	99 83	c	107 92	k	115 A2	s	123 C0	{
4	68 C4	D	76 D3	L	84 E3	T	92 E0	\	100 84	d	108 93	l	116 A3	t	124 4F	|
5	69 C5	E	77 D4	M	85 E4	U	93 9D	]	101 85	e	109 94	m	117 A4	u	125 D0	}
6	70 C6	F	78 D5	N	86 E5	V	94 5F	↑ or ^	102 86	f	110 95	n	118 A5	v	126 A1	~ (TILDE)
7	71 C7	G	79 D6	O	87 E6	W	95 6D	← or _	103 87	g	111 96	o	119 A6	w	127 07	DEL (RUBOUT)

SD-00217 Character code in octal at top and left of charts.

↑ means CONTROL

End of Appendix

Appendix B
Pseudo-Op Summary

The following table lists and describes the AOS MASM and AOS/VS MASM16 pseudo-ops. Each entry indicates whether a pseudo-op may function as an assembler directive and/or a value symbol. For more information, refer to the following sections of this manual:

- Chapter 7 contains a complete description of each pseudo-op
- Chapter 6 describes the various categories of pseudo-ops
- "Permanent Symbols" and "Pseudo-Ops" in Chapter 2 describe the properties of pseudo-op mnemonics and also explain the difference between assembler directives and value symbols
- "Symbol Interpretation" in Chapter 3 explains how the Macroassembler resolves the pseudo-ops that appear in your program

Table B-1. Pseudo-Op Summary

Pseudo-Op	Assembler Directive	Value Symbol	Most Common Use
(.) period	NO	YES	Return the value of the current location counter
.ARGCT	NO	YES	Return the number of arguments passed to a macro
.ASYM	YES	NO	Define an accumulating symbol
.BLK	YES	NO	Reserve a block of memory locations
.COMM	YES	NO	Reserve a labeled common area for intermodule communication
.CSIZ	YES	NO	Reserve an unlabeled common area for intermodule communication
.DALC	YES	NO	Define an ALC instruction or expression

(continues)

Table B-1. Pseudo-Op Summary

Pseudo-Op	Assembler Directive	Value Symbol	Most Common Use
.DCMR	YES	NO	Define a commercial memory reference instruction or expression
.DEMR	YES	NO	Define an extended memory reference instruction or expression, without accumulator
.DERA	YES	NO	Define an extended memory reference instruction that requires an accumulator
.DEUR	YES	NO	Define an extended user instruction or expression
.DFLM	YES	NO	Define a floating-point load or store instruction or expression that requires an accumulator
.DFLS	YES	NO	Define a floating-point load or store status instruction or expression that requires no accumulator
.DIAC	YES	NO	Define an I/O instruction requiring an accumulator
.DICD	YES	NO	Define an instruction requiring an accumulator and a count
.DIMM	YES	NO	Define an immediate-reference instruction requiring an accumulator
.DIO	YES	NO	Define an I/O instruction that does not use an accumulator
.DIOA	YES	NO	Define an I/O instruction that requires two fields
.DISD	YES	NO	Define an instruction with source and destination accumulators and no skip

(continued)

Table B-1. Pseudo-Op Summary

Pseudo-Op	Assembler Directive	Value Symbol	Most Common Use
.DISS	YES	NO	Define an instruction with source and destination accumulators with skip
.DMR	YES	NO	Define a memory reference instruction with displacement and index
.DMRA	YES	NO	Define a memory reference instruction with two or three fields
.DO	YES	NO	Assemble the following source code a specified number of times
.DUSR	YES	NO	Define a user symbol that will appear with the instruction symbols in the cross-reference listing
.DXOP	YES	NO	Define an instruction with source, destination, and operation fields
.EJEC	YES	NO	Begin a new listing page
.END	YES	NO	End-of-program indicator
.ENDC	YES	NO	Define the end of repetitive or conditional assembly lines
.ENT	YES	NO	Define one or more program entry points
.ENTO	YES	NO	Define an overlay entry
.EOF	YES	NO	Explicit end-of-file
.EXTD	YES	NO	Define one or more external displacement references (external symbol value is 8 bits or less)

(continued)

Table B-1. Pseudo-Op Summary

Pseudo-Op	Assembler Directive	Value Symbol	Most Common Use
.EXTN	YES	NO	Define one or more external normal references (external symbol value is 16 bits or less)
.EXTU	YES	NO	Treat undefined symbols as external displacements
.FORC	YES	NO	Force Link to include this library module in the program file
.GADD	YES	NO	Assign an expression value to a symbol
.GLOC	YES	NO	Initialize data fields relative to an external symbol
.GOTO	YES	NO	Suppress assembly of source lines until the specified symbol is found
.GREF	YES	NO	Assign an expression value to a symbol without affecting the sign bit
.IFE	YES	NO	Assemble the following source lines only if the value of the supplied expression equals zero
.IFG	YES	NO	Assemble the following source line only if the value of the supplied expression exceeds zero
.IFL	YES	NO	Assemble the following source lines only if the value of the supplied expression is less than zero
.IFN	YES	NO	Assemble the following source lines only if the value of the supplied expression does not equal zero
.LMIT	YES	NO	Specifies partial binding of an object file

(continued)

Table B-1. Pseudo-Op Summary

Pseudo-Op	Assembler Directive	Value Symbol	Most Common Use
.LOC	YES	YES	Set the current location counter
.LPOOL	YES	NO	Dump the currently defined literals into a data block
.MACRO	YES	NO	Define a macro
.MCALL	NO	YES	Indicate whether a macro has been called on the current assembly pass
.NLIT	YES	NO	Assign literals to NREL instead of ZREL
.NOCON	YES	YES	Enable or suppress the listing of conditional source lines
.NOLOC	YES	YES	Enable or suppress the listing of source lines that lack location fields
.NOMAC	YES	YES	Enable or suppress the listing of macro expansions
.NREL	YES	NO	Specify a predefined normal relocatable memory partition
.OB	YES	NO	Name an object file
.PASS	NO	YES	Return a value corresponding to the current assembly pass number
.PENT	YES	NO	Define a procedure entry
.POP	YES	YES	Return the value and relocation property of the last expression pushed onto the assembler stack and remove (pop) this information from the stack

(continued)

Table B-1. Pseudo-Op Summary

Pseudo-Op	Assembler Directive	Value Symbol	Most Common Use
.PTARG	YES	NO	Generate a procedure description for ?RCALL, ?KCALL, or ?RCHAIN
.PUSH	YES	NO	Push the value and relocation property of an expression onto the assembler stack
.RDX	YES	YES	Set the radix (base) for numeric input conversion
.RDXO	YES	YES	Set the radix (base) for numeric output conversion
.REV	YES	NO	Assign two revision-level numbers to a program file
.TITL	YES	NO	Assign a name to a listing header
.TOP	NO	YES	Returns the value and relocation property of the last expression pushed onto the stack; does not pop the information from the stack
.TSK	YES	NO	Specify the number of tasks in your program
.TXT	YES	NO	Store the octal equivalent of an ASCII text string in consecutive memory words
.TXTE	YES	NO	Set the leftmost bit for even byte parity
.TXTF	YES	NO	Set the leftmost bit to one unconditionally
.TXTM	YES	YES	Specify left/right or right/left bytepacking within words
.TXTN	YES	YES	Terminate an even length byte string with no null bytes or two null bytes

(continued)

Table B-1. Pseudo-Op Summary

Pseudo-Op	Assembler Directive	Value Symbol	Most Common Use
.TXTO	YES	NO	Set the leftmost bit for odd byte parity
.XPNG	YES	NO	Delete symbol and macro definitions from the current assembly
.ZREL	YES	NO	Specify the predefined lower page zero relocatable memory partition

(concluded)

End of Appendix

Appendix C
Assembly Error Codes

Macroassembler error messages appear as single letter codes in the first three character positions of a listing line. The first error code appears in character position three of the line where the error occurred. If there is a second error, the code is output in position two. The error code for a third error appears as the first character of the listing line.

Assembler errors are output as part of the assembly listing. They are also sent to the output file. If the listing is suppressed, the error listing is sent to the output file only. If there is a listing device, output of errors to the output file can be suppressed. Certain errors encountered on the first pass will be output since the Macroassembler may not detect them on its second pass.

The list of Macroassembler error codes is as follows:

A	Address error
B	Bad character
C	Macro error
D	Radix error
E	Equivalence error
F	Format error
G	Global error
I	Parity error on input
K	Conditional or repetitive assembly error
L	Location counter error
M	Multiply-defined symbol error
N	Number error
O	Overflow error or stack error
P	Phase error
Q	Questionable line
R	Relocation error
U	Undefined symbol error
V	Variable label error
X	Text error

We describe each of these error codes in the following sections. Each section has a sample of assembly-language source code that would cause the error. These samples may help you find an error in your source code. However, there is no way to pinpoint all possible causes of assembly errors.

Addressing Error (A)

An addressing (A) error indicates an illegal address in a memory reference instruction (MRI). For example:

1. A page zero relocatable instruction references a normal relocatable (NREL) address. For example:

```
        .NREL 0
G:      10          ;G has a NREL address
        .ZREL
        STA 0,G     ;G cannot be used in a ZREL instruction
```

2. An NREL address references an address outside the location counter's relative address range: (.-200≤ displacement ≤ .+177). For example:

```
        .NREL
        LDA 0,Y      ;Y is outside the instruction's range
        .LOC .+416
Y:      2
```

Bad Character (B)

Error code B indicates an illegal character in a symbol. The line containing a symbol that has an illegal character will be flagged with a B. A bad character error often causes other errors. For example:

```
        .NREL
.A%:    LDA 1,23        ;% in label symbol is illegal
```

Macro Error (C)

The macro error (C) occurs under the following conditions:

1. If you try to continue the definition of a macro when it is not the last macro defined. For example:

```
.MACRO□A
macro-definition%
.
.
.
.MACRO□A
Macro-definition%      ;Legal continuation
.
.
.MACRO□B
macro-definition%
.
.
.
.MACRO□A                          ;Illegal to continue any
                                 ;except macro B
```

2. If a macro exhausts the Macroassembler's working space. However, this should only occur if the macro definition causes endless recursion.

3. If more than 63_{10} arguments are specified.

4. If you try to build a permanent symbol table (MASM.PS) with an existing MASM.PS in your directory, or on your searchlist.

Radix (D)

Error code D occurs on a .RDX or .RDXO pseudo-op:

1. when .RDX contains an expression that is not in the range 2-20;

2. or when .RDXO contains an expression that is not in the range 8-20;

3. or when you used a digit that is not within the current input radix.

For example:

```
        .RDX   4*6        ;Illegal expression (out-of range)
        .END

        .RDX   2
B:      35                ;B is outside the current radix
```

Equivalence Error (E)

Error code E occurs when an equivalence line contains an undefined symbol on the righthand side of the equals sign. This error may occur on pass one, before the symbol on the righthand side has been defined. It can occur on pass two if the symbol is never defined. For example:

```
        A=B             ;Pass one: B is undefined

        .NREL
        A=B             ;Pass two: B is still undefined
        .END
```

Format Error (F)

A format (F) error results from any attempt to use a format that is not legal for the type of line. The format error often occurs in conjunction with other errors.

When a format error occurs in an instruction, the code generated by the instruction reflects only those fields assembled <u>before</u> the error was detected. For example:

```
        ADD 2              ;Not enough operands

        STA 0,10,3,SNC     ;Too many operands and wrong
                           ;operand for instruction type

        .ZREL-1            ;ZREL cannot have an argument

        .DUSR C = DIAS 0,PTR       ;Attempt to give an
                                   ;argument to a symbol
                                   ;defined in a .DUSR pseudo-op
```

External/Internal Symbol Error (G)

An external/internal symbol error occurs when there is an error in the declaration of an external or entry symbol. For example:

```
        .ENT   HH          ;HH is not defined
        .END

AA:                        ;AA is an entry in a program in which
        .EXTN AA           ;the symbol is declared as an external
        .END
```

Input (Parity) Error (I)

An input parity (I) error occurs when an input character does not have even parity. The Macroassembler substitutes a back slash (\) for any incorrect character. It also flags the line containing the error with an I.

Conditional Assembly Error (K)

A conditional assembly (K) error occurs wnen an .ENDC pseudo-op is not preceded by a .DO or .IFx pseudo-op. For example:

```
.DO 2
  .
  .
  .
.ENDC
.ENDC    ;This .ENDC does not have a corresponding .DO
         ;or IFx pseudo-op
```

Location Error (L)

The location (L) error code occurs when the Macroassembler finds an error that affects the location counter. For example:

1. The expression in a .LOC evaluates to less than zero, or cannot be evaluated on the Macroassembler's first pass. If the expression is outside the range of locations or cannot be evaluated, the .LOC is ignored, and the location counter is unchanged.

   ```
   .LOC    -1      ;Illegal expression
                   ;(less than zero)
   ```

2. The expression in a .BLK statement cannot be evaluated on the first pass of the Macroassembler, or its value, when added to the current value of the program location counter, is less than zero. If an L error occurs, the .BLK statement is ignored and the location counter is changed.

   ```
   A:      0
           .BLK .+100
   ```

Multiple Definition Error (M)

The multiple definition (M) code flags a multiply-defined symbol. For example, a symbol that appears as a label cannot be redefined as another unique label. Any multiply-defined symbol will be flagged with an M each time the symbol appears. For example:

```
        .NREL
A:      0
A:      1        ;A cannot be redefined
```

The second definition of A will also be flagged as a phase error (P) on the second assembler pass (see Phase Error).

Number Error (N)

The number (N) code is issued when a number exceeds the proper storage limitations for the type of number. The N error occurs under the following conditions:

1. An integer is greater than or equal to 2^{16}. The number is evaluated modulo 2^{16}. For example:

   ```
   .RDX 10
   65539   ;Integer value is out-of-range
   ```

2. A double-precision integer is greater than or equal to 2^{32}. The number is evaluated modulo 2^{32}.

3. A floating point number is larger than $7.2*10^{75}$.

Field Overflow Error (O)

A field overflow (O) error occurs when:

1. variable stack space is exceeded;

2. a .TOP or .POP is given with no previous .PUSH; or

3. when an instruction operand is not within the required limits (e.g., 0-3 for an accumulator, 0-7 for a skip field, etc). When overflow occurs in an instruction field, such as an accumulator field, the field will remain unchanged.

For example:

```
LDA 5,.-3        ;5 is illegal in the
                 ;instruction's accumulator field

.DIAC R=14000
R 1
```

Phase Error (P)

A phase (P) error occurs when the Macroassembler finds, on pass two, an unexpected difference from the source program scan on pass one. For example, a symbol defined on the first pass that has a different value on the second pass will cause a phase error. If (as in the following example) a symbol is multiply-defined, the M error flags each statement containing the symbol. The phase error flags the second and any subsequent attempt to redefine the symbol. For example:

```
        .NREL

B:      0
B:      0       ;This line will receive a P and an M
                ;error flag
```

Questionable Line (Q)

A questionable (Q) error occurs when you have:

1. used a # or @ atom improperly,
2. used a ZREL value where an absolute value is expected, or
3. used an instruction that may cause a skip immediately before a two-word instruction.

For example:

```
        ADD    0,#2      ;ALC instruction with a # atom
                         ;requires a skip mnemonic

        .ZREL
FLD:    .BLK 10
        .NREL
        LDA    0,FLD,2  ;Assembler expects an absolute value
        .END             ;for FLD


        MOV    1,1,SNR  ;Instruction with a skip field
        ELDA   0,SYMB   ;precedes a two-word instruction


        MOV#   0,1       ;The instruction's no load bit is
                         ;set, but no skip condition is specified
```

Relocation Error (R)

The relocation (R) error occurs when an expression cannot be evaluated to a legal relocation type (absolute, relocatable, or byte-relocatable as described in Chapter 3). A relocation error also occurs when an expression mixes ZREL and unshared NREL symbols, or NREL and shared ZREL symbols. For example:

```
        .NREL
E:      10      ;Contents absolute.
E+E             ;Contents NREL byte.
E+E+E           ;Illegal--contents not absolute, relocatable,
                ;or byte relocatable.
```

Undefined Symbol Error (U)

The undefined symbol (U) error occurs on pass two, when the assembler encounters a symbol whose value was not known on pass one. The error also occurs on pass one when the definition of a symbol (by equivalence) depends upon another symbol whose value is unknown at that point. For example:

```
        LDA  2,B        ;Causes a U error if B is undefined
```

See also the example given for equivalence error E.

Variable Label Error (V)

A variable label (V) error occurs if anything other than a symbol follows the .GOTO pseudo-op. For example:

```
        .GOTO 14        ;14 is not a symbol
```

Text Error (X)

An error occurring in a string is flagged as a text error (X). A text error occurs if the expression delimiters < and > within a string do not enclose a recognizable arithmetic or logical expression. Relational expressions cannot be used within text strings. For example:

```
        .TXT #<X+ Y>#   ;Spaces are illegal in expressions

        .TXT #<+>#      ;Expressions must have operands

        .TXT #<X=>Y>#   ;Relational operator (=) is illegal
```

End of Appendix

Appendix D
Assembling 16-Bit Programs with AOS/VS MASM

In Chapter 1, we described how you may assemble, link, and run 16-bit ECLIPSE assembly language programs on 32-bit ECLIPSE computers. The AOS/VS MASM16 utility assembles your AOS source files to produce AOS/VS-compatible object files. However, you must use the AOS/VS Macroassembler if you want to take full advantage of the 32-bit ECLIPSE computer's performance. Therefore, you may want to convert your 16-bit programs into a format that is compatible with the AOS/VS Macroassembler. This appendix outlines some of the conventions you should be aware of when you convert 16-bit programs.

This information is provided as a programming aid. You should refer to the AOS/VS Macroassembler Reference Manual for more information about AOS/VS MASM conventions. The ECLIPSE MV/8000® Principles of Operation manual describes the ECLIPSE MV/8000 instruction set and programming environment. It also explains the differences between the 16-bit and 32-bit instruction sets.

AOS-Only Pseudo-Ops

You should not use any of the following AOS MASM and AOS/VS MASM16 pseudo-ops if you want to assemble your program with the AOS/VS Macroassembler. The AOS/VS Macroassembler will generate an error if it detects any of these pseudo-ops.

.ASYM
.DALC
.DCMR
.DEMR
.DERA
.DEUR
.DFLM
.DFLS
.DIAC
.DICD
.DIMM
.DIO
.DIOA
.DISD
.DISS
.DMR
.DMRA
.DXOP
.ENTO
.EXTU
.GADD
.GOTO
.GREF
.LMIT
.LPOOL
.NLIT
.PENT
.PTARG
.TOP

Pseudo-Op Interpretation

The AOS and AOS/VS Macroassemblers interpret some pseudo-ops differently. The following list outlines these differences.

* The .LOC pseudo-op is interpreted differently when it is pushed and subsequently popped from the stack. AOS MASM and AOS/VS MASM16 returns the location counter's relocation

base when .LOC is popped from the stack. The AOS/VS Macroassembler returns both the value and relocation base of the location counter.

- Arguments to the .NREL pseudo-op are treated differently. The AOS Macroassembler recognizes an .NREL expression as either zero or non-zero. The AOS/VS Macroassembler will generate an error if an .NREL expression does not evaluate to 0, 1, 4, 5, 6, or 7.

Macros

The AOS and AOS/VS Macroassemblers handle macros differently in several areas. Refer to Chapter 5 for a description of the AOS Macroassembler's macro features.

- You cannot pass a carriage return, form feed, or NEW LINE character as arguments to an AOS MASM or AOS/VS MASM16 macro. AOS/VS MASM adds the space, horizontal tab, comma, left bracket ([), and right bracket (]) to this list of restricted characters.

- The AOS/VS Macroassembler requires that the macro definition terminator (the % character) be the only character on the last line of a macro definition.

- AOS MASM and AOS/VS MASM16 allows you to use the dollar sign ($) character to generate unique labels within macros. The AOS/VS Macroassembler does not support this feature.

- The AOS/VS Macroassembler does not allow you to stop and restart a macro's definition. The AOS/VS Macroassembler generates an error if you use the following construction.

```
.MACRO TEST      ;DEFINE MACRO CALLED TEST
10
20
%                ;TERMINATE TEST'S DEFINITION
LDA 0,0
ADD 0,1
.MACRO TEST      ;CONTINUE TEST'S DEFINITION
30               ;(ILLEGAL FOR AOS/VS MASM PROGRAMS)
40
%
```

Double-Precision Indicator (D)

AOS MASM and AOS/VS MASM16 uses D to indicate that a data item is to be assembled into two, 16-bit words (a double-precision integer). The AOS/VS Macroassembler assembles data into two 16-bit words by default. It does not recognize D as a double-precision indicator. The AOS/VS Macroassembler provides pseudo-ops that allow you to store data in single (16 bit) words or double (32 bit) words. See the description of the .ENABLE pseudo-op in Chapter 7 of the AOS/VS Macroassembler Reference Manual for more information.

Literal Values in Memory Reference Instructions

The 32-bit instruction set has provisions for placing immediate data in instructions. Therefore, the AOS/VS Macroassembler does not support the use of literals in memory reference instructions.

Index Mode Default Conditions

The AOS/VS Macroassembler defaults to program counter relative addressing (index value of 1) if an instruction has no index mode specified. AOS/VS MASM allows you to change this default setting with the .ENABLE pseudo-op.

CLI Command-Line Switches

Some of the AOS and AOS/VS MASM CLI function and argument switches are different. Therefore, you may have to modify your AOS CLI macros before you use them with the AOS/VS Macroassembler.

End of Appendix

Index

. 7-3

2-37

** 2-37, 4-7

.ARGCT 7-4
.ASYM 7-6
.BLK 7-7
.COMM 7-8
.CSIZ 7-10
.DALC 7-11
.DCMR 7-13
.DEMR 7-15
.DERA 7-17
.DEUR 7-19
.DFLM 7-21
.DFLS 7-23
.DIAC 7-25
.DICD 7-26
.DIMM 7-28
.DIO 7-30
.DIOA 7-32
.DISD 7-34
.DISS 7-36
.DMR 7-38
.DMRA 7-40
.DO 7-42
.DUSR 7-45
.DXOP 7-47
.EJEC 7-49
.END 7-51
.ENDC 7-52
.ENT 7-54
.ENTO 7-56
.EOF 7-58
.EXTD 7-60
.EXTN 7-62
.EXTU 7-64
.FORC 7-65
.GADD 7-66
.GLOC 7-67
.GOTO 7-69
.GREF 7-70
.IFE 7-71
.IFG 7-71
.IFL 7-71
.IFN 7-71
.LMIT 7-73
.LOC 7-75
.LPOOL 7-77
.MACRO 7-79
.MCALL 7-81
.NLIT 7-83
.NOCON 7-84
.NOLOC 7-86
.NOMAC 7-88
.NREL 7-90
.OB 7-92
.PASS 7-94
.PENT 7-95
.POP 7-97
.PTARG 7-98
.PUSH 7-99
.RDX 7-100
.RDXO 7-103
.REV 7-105
.TITL 7-107
.TOP 7-108
.TSK 7-109
.TXT 7-110
.TXTE 7-110
.TXTF 7-110
.TXTM 7-113
.TXTN 7-115
.TXTO 7-110
.XPNG 7-117
.ZREL 7-119

@ 2-36
@OUTPUT 1-10

A

ABS 7-119
Absolute 3-29
  addresses 3-20
  addressing 3-29
  expression 3-20
  partition 3-12
  values 3-20

AC relative addressing 3-29
Address space, logical 3-9
Addresses
  absolute 3-20
  relocatable 7-119
Addressing 3-29
  absolute 3-29
  AC relative 3-29
AOS/VS MASM16 1-4
Apostrophe 2-15
Argument
  formal 5-4
  index 3-29
  mode 3-29
  null 5-8
Argument switch 8-2
ASCII character set A-1
Assembler
  directives 2-22, 2-45, 6-1
  stack 7-97, 7-99
  value symbols 6-1
Assembly
  conditional 6-3, 7-52, 7-71
  error codes C-1
  listing 4-2
  listing sample 4-3
  listings, macro expansion in 5-10
  pass 7-94
  process 3-1
  repetitive 6-3, 7-42, 7-52
Assembly language instructions 2-42
Assigning locations 3-9
Assignment statement 2-47
Asterisks (**) 2-37, 4-7
At sign (@) 2-36
Atom, definition of 2-6

B

Bases,relocation 3-15, 3-18, 3-20
Binary operators 2-30
Bit vi
  fields vi
  least significant vi
  most significant vi
Box vi, 2-8
Byte-pointers 3-24
Byte-relocation 3-24

C

Carriage Return 2-7
Character set 2-1
Coding aids 7-1
Comments 2-41
Common area 7-8, 7-10
Conditional assembly 6-3, 7-52, 7-71
Constants, single-precision floating-point 2-16
Cross-reference listing 4-8
Cross-referencing 7-45
Current location counter 7-3

D

Data
  fields 7-67
  statement 2-40
Debugging 1-2
Delete symbol and macro definitions 7-117
Delimiters 2-7
Digit representation 2-10
Displacement 3-29
  external 7-60

E

End-Of-File 7-58
End-of-line characters 2-7
End-Of-Program indicator 7-51
Error codes C-1
Error listing 4-10
Exponent 2-17
Expression 2-28
  absolute 3-21
  byte-relocatable 3-24
  definition of 2-7
  relocatable 3-23
External
  displacement 7-60
  entry 7-54
  normal references 7-62

F

File termination 6-3
Filename extensions 8-6
Filenames 8-6
Floating-point constants 2-16
Force link a module 7-65
Form feed 2-7
Formal arguments 5-4
Function switch 8-1

G

General references 7-1
Generated numbers and symbols 5-1, 5-20

I

Index 3-29
Indirect addressing bit 2-36
Input, numeric 7-100
Instruction symbols 2-24
- redefining 2-25

Integer-generating formats 2-14
Integers
- double-precision 2-12
- single-precision 2-9

Intermodule communication 6-5

L

Labels 2-39
- generated 5-19

Least significant bit vi
Library 7-65
Link utility 3-17
Linking 1-2
Listing 7-84, 7-88
- assembly 4-2
- control 6-9
- control pseudo-ops 4-7
- error 4-10
- page 7-49

Literals 3-34, 7-77, 7-83
Location counter 3-10, 6-2, 7-75
Logical address space 3-9
Loops and conditionals in macros 5-12
Lower page zero 3-9, 7-119

M

Machine language 1-1
Macro 2-43, 5-1, 6-5, 7-4, 7-79
- calls 5-5
- definition of 5-1
- examples 5-14
- expanding 3-8
- expansion in assembly listings 5-10
- loops and conditionals in 5-12
- null arguments 5-8
- number of arguments 5-5
- processing 3-8
- special characters 5-8

Macroassembler
- command line 8-2
- input to 2-1
- introduction to 1-1
- operating procedures 8-1
- output from 4-1
- simplest use of 1-7
- symbol tables 8-9

Mantissa 2-17
MASM16 1-4
Memory 3-9
- assigning locations 3-9
- management 6-2
- normal relocatable 3-9
- page zero relocatable 3-9
- partition 7-90, 7-119
- reserve a block of 7-7

Memory reference instruction 3-29
Mode 3-29
Most significant bit vi
MRI, see Memory reference instruction

N

NEW LINE 2-7
No-load bit 2-37
Normal, relocation 7-90
Normalized 2-17
NREL partition 3-14, 7-90
Null
  arguments 5-8
  word 7-111
Number sign (#) 2-37
Numbers 2-9
Numeric
  input 7-100
  output 7-103

O

Object file 1-2, 4-1, 7-92
Operating procedures 8-1
  macroassembler 8-1
Operator 2-29
  B 2-33
  binary 2-29
  bit alignment 2-32
  logical 2-31
  priority of 2-34
  relational 2-31
  unary 2-30
Output, numeric 7-103

P

Pages 3-9
Partition 3-10
  absolute 3-12
  NREL 7-90
  NREL shared 3-14
  NREL unshared 3-14
  ZREL 3-14, 7-119
PC relative addressing 3-29
Permanent symbol table 8-10
  building 8-11
  size 8-12
Permanent symbols 2-22
Pseudo-op
  definition of 1-7
  summary B-1
Pseudo-ops 2-22, 2-45, 7-1
  conditional assembly 6-3
  descriptions 7-1
  file termination 6-3
  general references 7-1
  intermodule communication 6-5
  listing control 4-7, 6-9
  location counter 6-2
  macro-related 5-12
  macros 6-5
  memory management 6-2
  radix control 6-11
  repetitive assembly 6-3
  stack control 6-10
  symbol deletion 6-16
  symbol table 6-12
  text strings 6-11
  types of 6-1

Q

Quotation mark 2-14

R

Radix 7-100, 7-103
  control 6-11
Redefining instruction symbols 2-25
Relocatability 3-15
Relocatable addresses 7-119
Relocation
  bases 3-18, 3-20
  normal 7-90
  symbols 3-18
Repetitive assembly 6-3, 6-9, 7-42, 7-52
Revision level 7-105

S

Sample assembly listing 4-3
Single-precision floating-point constants 2-16
Special
  atoms 2-36
  characters 5-8

Stack 7-97, 7-99
  control 6-10
Statement
  body 2-41
  components 2-6
  format 2-38
  types 2-41
Strings
  text 6-11, 7-110, 7-113, 7-115
Switch
  argument 8-2
  function 8-2
Symbol 2-19
  deletion 6-16
  global 2-27
  instruction 2-24
  interpretation 3-1
  length 8-12
  local 2-27
  macro 2-26
  names 2-20
  redefining, macro 2-26
  redefining, user 2-27
  relocation 3-18
  reserved 2-27
  resolution 8-9
  semipermanent 2-24
  types 2-21
  user 2-27
Symbol table 8-9
  building a permanent 8-11
  macroassembler 8-9
Symbol table pseudo-op, format 6-15
Syntax checking 3-4
System calls 2-43, 5-18
  expanding 3-8
  processing 3-8

T

Table
  building a permanent symbol 8-11
  macroassembler symbol 8-9
  permanent symbol 3-2
  temporary symbol 3-2
Tasks 7-109
Terminators 2-7
Text strings 6-11, 7-110, 7-113, 7-115

U

Unary operators 2-30
User symbol 7-45

V

Value symbol 2-23
Values, absolute 3-20

W

Word, 16-bit vi

Z

ZREL partition 3-14, 7-90

Data General

users group

Installation Membership Form

Name ______________________ Position ______________________ Date __________

Company, Organization or School ____

Address ______________________ City ______________ State __________ Zip __________

Telephone: Area Code __________ No. ______________ Ext. ______________

CUT ALONG DOTTED LINE

1. Account Category

□ OEM
□ End User
□ System House
□ Government

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
MV/Series ECLIPSE®		
Commercial ECLIPSE		
Scientific ECLIPSE		
Array Processors		
CS Series		
NOVA® 4 Family		
Other NOVAs		
microNOVA® Family		
MPT Family		

Other __________
(Specify) __________
____

3. Software

□ AOS
□ AOS/VS
□ AOS/RT32
□ MP/OS
□ MP/AOS
□ RDOS
□ DOS
□ RTOS
□ Other

Specify __________

4. Languages

□ ALGOL
□ DG/L
□ COBOL
□ Interactive COBOL
□ PASCAL
□ Business BASIC
□ BASIC
□ Assembler
□ FORTRAN 77
□ FORTRAN 5
□ RPG II
□ PL/1
□ APL
□ Other

Specify __________
____

5. Mode of Operation

□ Batch (Central)
□ Batch (Via RJE)
□ On-Line Interactive

6. Communication

□ HASP
□ HASP II
□ RJE80
□ RCX 70
□ RSTCP
□ 4025
□ X.25
□ SAM
□ CAM
□ XODIAC™
□ DG/SNA
□ 3270
□ Other

Specify __________
____

7. Application Description

○ __________
____
____
____

8. Purchase

From whom was your machine(s) purchased?

□ **Data General Corp.**
□ Other
Specify __________
____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ __________
____
____

Data General

Data General Corporation, Westboro, Massachusetts 01580, (617) 366-8911

FOLD *FOLD*

TAPE *TAPE*

FOLD *FOLD*

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

DataGeneral
ATTN: Users Group Coordinator (C-228)
4400 Computer Drive
Westboro, MA 01581

Data General

TP__________

TIPS ORDER FORM
Technical Information & Publications Service

BILL TO:
COMPANY NAME____________________
ADDRESS ____________________
CITY ____________________
STATE__________ ZIP __________
ATTN: ____________________

SHIP TO: (if different)
COMPANY NAME____________________
ADDRESS ____________________
CITY ____________________
STATE__________ ZIP __________
ATTN: ____________________

QTY	MODEL #	DESCRIPTION	UNIT PRICE	LINE DISC	TOTAL PRICE

(Additional items can be included on second order form)

[Minimum order is $50.00]

Tax Exempt #__________
or Sales Tax (if applicable)

TOTAL	
Sales Tax	
Shipping	
TOTAL	

METHOD OF PAYMENT

□ Check or money order enclosed
For orders less than $100.00

□ Charge my □ Visa □ MasterCard
Acc't No.__________ Expiration Date__________

□ Purchase Order Number:__________

SHIP VIA

□ DGC will select best way (U.P.S or Postal)

□ Other:
□ U.P.S. Blue Label
□ Air Freight
□ Other ____________________

NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.

Person to contact about this order ____________________ Phone __________ Extension ______

Mail Orders to:

Data General Corporation
Attn: Educational Services/TIPS F019
4400 Computer Drive
Westboro, MA 01580
Tel. (617) 366-8911 ext. 4032

Buyer's Authorized Signature Date
(agrees to terms & conditions on reverse side)

Title

DGC Sales Representative (If Known) Badge #

**DISCOUNTS APPLY TO
MAIL ORDERS ONLY**

CUT ALONG DOTTED LINE

educational services

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. **PRICES**
   Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. **PAYMENT**
   Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. **SHIPMENT**
   Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4. **TERM**
   Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. **CUSTOMER CERTIFICATION**
   Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. **DATA AND PROPRIETARY RIGHTS**
   Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

7. **DISCLAIMER OF WARRANTY**
   DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. **LIMITATIONS OF LIABILITY**
   IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. **GENERAL**
   A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.

Data General

TIPS ORDERING PROCEDURE:

Technical literature may be ordered through the Customer Education Service's Technical Information and Publications Service (TIPS).

1. Turn to the TIPS Order Form.
2. Fill in the requested information. If you need more space to list the items you are ordering, use an additional form. Transfer the subtotal from any additional sheet to the space marked "subtotal" on the form.
3. Do not forget to include your MAIL ORDER ONLY discount. (See discount schedules on the back of the TIPS Order Form.)
4. Total your order. (MINIMUM ORDER/CHARGE after discounts of $50.00.)

   If your order totals less than 100.00, enclose a certified check or money order for the total (include sales tax, or your tax exempt number, if applicable) plus $5.00 for shipping and handling.
5. Please indicate on the Order Form if you have any special shipping requirements. Unless specified, orders are normally shipped U.P.S.
6. Read carefully the terms and conditions of the TIPS program on the reverse side of the Order Form.
7. Sign on the line provided on the form and enclose with payment. Mail to:

   TIPS
   Educational Services – M.S. F019
   Data General Corporation
   4400 Computer Drive
   Westboro, MA 01580
8. We'll take care of the rest!

educational services

User Documentation Remarks Form

Your Name ______________________________ Your Title ______________________

Company ____

Street ____

City ______________________________________ State ____________ Zip __________

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title ______________________________________ Manual No. ______________

Who are you? □EDP Manager □Analyst/Programmer □Other ______________

□Senior Systems Analyst □Operator ______________

What programming language(s) do you use? ______________________________________

How do you use this manual? *(List in order: 1 = Primary Use)* ______________________

___ Introduction to the product ___ Tutorial Text ___ Other

___ Reference ___ Operating Guide ______________

About the manual:		Yes	Somewhat	No
	Is it easy to read?	□	□	□
	Is it easy to understand?	□	□	□
	Are the topics logically organized?	□	□	□
	Is the technical information accurate?	□	□	□
	Can you easily find what you want?	□	□	□
	Does it tell you everything you need to know	□	□	□
	Do the illustrations help you?	□	□	□

If you have any comments on the software itself, please contact Data General Systems Engineering.
If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Remarks:

Date

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

POSTAGE WILL BE PAID BY ADDRESSEE

DataGeneral
User Documentation, M.S. E-111
4400 Computer Drive
Westborough, Massachusetts 01581

User Documentation Remarks Form

Your Name ______________________ Your Title ______________

Company ______________________________

Street ______________________________

City ______________________ State __________ Zip __________

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title ______________________ Manual No. __________

Who are you? □EDP Manager □Analyst/Programmer □Other __________
□Senior Systems Analyst □Operator __________

What programming language(s) do you use? ______________________

How do you use this manual? *(List in order: 1 = Primary Use)* ______________

___ Introduction to the product ___ Tutorial Text ___ Other
___ Reference ___ Operating Guide __________

		Yes	Somewhat	No
About the manual:	Is it easy to read?	□	□	□
	Is it easy to understand?	□	□	□
	Are the topics logically organized?	□	□	□
	Is the technical information accurate?	□	□	□
	Can you easily find what you want?	□	□	□
	Does it tell you everything you need to know	□	□	□
	Do the illustrations help you?	□	□	□

If you have any comments on the software itself, please contact Data General Systems Engineering.
If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Remarks:

Date

DataGeneral

User Documentation, M.S. E-111
4400 Computer Drive
Westborough, Massachusetts 01581

Data General Corporation, Westboro, MA 01580