DataGeneral

Customer Documentation

AOS/VS Macroassembler (MASM) Reference Manual

AOS/VS Macroassembler (MASM) Reference Manual

093-000242-02

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. 093-000242

Printed in the United States of America
Revision 02, August 1987

NOTICE

AOS/VS Macroassembler (MASM)
Reference Manual
093-000242-02

Revision History:
Original Release - November 1980
First Revision - February 1984
Second Revision - August 1987

Effective with:
AOS/VS REV. 7.50

A vertical bar in the margin of a page indicates substantive change from the previous revision.

Preface

This manual describes the use and operation of the AOS/VS Macroassembler (MASM) utility. It assumes you have assembly language programming experience and are familiar with the ECLIPSE® MV/Family 32-bit assembly language instruction set.

We have written this manual as a reference tool. The following list summarizes the contents of each chapter and appendix.

Chapter 1 Introduces the AOS/VS 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 can use in your program.

Chapter 7 Provides detailed descriptions of the 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 MASM pseudo-ops.

Appendix C Lists and describes all assembly and command line errors.

Appendix D Lists incompatibilities between AOS MASM, AOS/VS MASM, and AOS/VS MASM16.

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:

- The *AOS/VS Debugger and File Editor User's Manual* (093-000246) 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/VS Link and Library File Editor (LFE) User's Manual* (093-000245) describes the Link utility in depth. After assembling your program, you must further process it with Link to produce an executable program file.
- *AOS/VS System Concepts* (093-000335) and *AOS/VS System Call Dictionary* (093-000241) document the AOS/VS system calls and system parameters. System calls are predefined macros that perform commonly used operations.
- *ECLIPSE® MV/Family 32-Bit Principles of Operation* (014-000648) documents all ECLIPSE® MV/Family 32-Bit assembly language instructions. This manual also describes concepts specific to the ECLIPSE® MV/Family 32-Bit hardware.

Reader, Please Note:

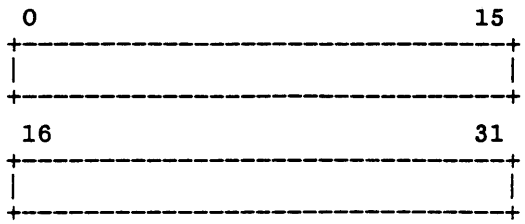
We use these conventions for command formats in this manual:

COMMAND□required<□*optional*> ...

Where	Means
COMMAND	You must enter the command as shown.
required	You must enter some argument. Sometimes, we use $\left\{ \begin{array}{l} \text{required}_1 \\ \text{required}_2 \end{array} \right\}$ 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.
□	The box represents any combination of spaces, horizontal tabs, and one comma. We generally use the box to separate a command and its arguments.
)	This close parenthesis represents the CLI prompt.

Additionally, we use the symbol ↵ 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 31 is the *least significant* bit in a 32-bit sequence.

Note that we divide bit fields into 16-bit quantities. Each 16-bit segment is called a *word*.

We represent sections of assembly language source code in the following font:

```
        A=2                     ;Initialize A and B.
        B=4
X:      1                       ;Location X contains a 1.
Y:      3                       ;Location Y contains a 3.
```

32- and 16-Bit Programs

32-bit programs include ECLIPSE MV/Family 32-Bit assembly language instructions and run under the 32-bit AOS/VS operating system. *16-bit programs* use 16-bit ECLIPSE instructions and are usually designed to run under the 16-bit AOS. You can run both 16- and 32-bit programs under AOS/VS if you assemble and link them correctly.

You should use the AOS/VS Macroassembler to assemble all new assembly language code, including both 16- and 32-bit programs. To assemble old AOS assembly language source code under AOS/VS, you should use MASM16.

While AOS/VS MASM is faster, more reliable, and has more features and better error checking, MASM16 has a few features that the newer assembler lacks. Older assembly language source code may be dependent on those features. This is the reason for using two different assemblers.

After assembling your 32- or 16-bit program, use the AOS/VS Link utility to produce a program file. That is, regardless of which Macroassembler you use (AOS or AOS/VS) you must always use the AOS/VS Link to produce an AOS/VS program file (i.e., a program that will run under AOS/VS). The *AOS/VS Link and Library File Editor (LFE) User's Manual* provides more information on linking 32- and 16-bit programs.

Contacting Data General

If you have comments on this manual, please use the prepaid Comment 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.

End of Preface

Contents

1 Introduction to the Macroassembler

2 Input to the Macroassembler

3 The Assembly Process

4 Output from the Macroassembler

5 Macros and Generated Numbers and Symbols

6 Types of Pseudo-Ops

Tables

Figures

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 branch instruction to address 377_8:

```
000377
```

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

Coding a program that utilizes the many ECLIPSE® MV/Family 32-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 branch statement is now

```
JMP             377
```

This instruction is much simpler to read because the symbol JMP is similar to the English word 'jump' and 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 readily understood by the computer. Thus, you must somehow 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 editing your program* – The first step in producing your program is, of course, entering the appropriate assembly language instructions 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 should 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 module(s) apart and rearranges them into an image of your program reflecting how 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 you program, you can execute it by typing

) XEQ program-filename ↵

   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

   - either cause a runtime error
   - or not perform the desired action(s)

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/VS 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 very 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 can

- 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
- expands macros and system calls (we describe these later in this chapter)

Macroassembler Output

The Macroassembler can produce four types of output:

- object file
- assembly and cross-reference listing
- error listing

- permanent symbol file

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 followed by their values. This listing is included as part of the assembly listing.

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

The *permanent symbol file* holds symbol definitions for use in future assemblies. If you use a permanent symbol file, you need not redefine frequently used symbols for each assembly. The AOS/VS software package provides a standard permanent symbol file for your use.

Special Features of MASM

As we have seen, 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 two different internal number representations at your disposal (integer, and floating-point).
Expression evaluation	The Macroassembler provides arithmetic, logical, and relational operators that you can 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 can direct MASM to assemble a series of source statements a specified number of times; that is, you can 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 can implement an IF-THEN-ELSE structure at assembly time.
16- and 32-bit data placement	The Macroassembler can allocate either 16 or 32 bits of memory for each data value in your source module.
Text string storage	You can direct MASM to store in memory the ASCII codes for any string of characters.
Radix control	You can alter the radix (base) for numeric input to and listing output from the Macroassembler.

Assembler stack	The Macroassembler provides a push-down stack that operates at assembly time.
Intermodule communication	You can 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.

The Macroassembler supplies these and other programming tools to aid you in program development. Note that the Link utility and the AOS/VS 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 want 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 does not execute 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 your source module:

- .TITLE
- a location directive
- .END

.TITLE places a name in the object module for later use by Link and the Library File Editor (LFE). 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 want 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 inform 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 shared memory. The pseudo-op statement that conveys this to the Macroassembler is

```
.NREL 1
```

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.

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 summary, 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/VS). Instead, you must use I/O system calls (predefined macros that handle input and output for you). *AOS/VS System Concepts* and *AOS/VS System Call Dictionary* describe the I/O system calls in detail.

Basic MASM Command Line

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

```
) XEQ MASM source-file ↵
```

where:

XEQ	is a CLI command that executes a program
MASM	is the name of the Macroassembler program (less the .PR extension)
source-file	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., **source-file**), 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**.

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

```
) XEQ MASM PROG1 ↵
```

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. The Macroassembler provides you with many options at assembly time. For example, you can

- produce an assembly listing
- send errors to a specific file (i.e., besides **@OUTPUT**)
- specify the name for your object file
- suppress production of the object file
- receive statistics about the assembly
- direct MASM to distinguish between uppercase and lowercase letters

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

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 naming conventions for files in more detail. That chapter also explains the Macroassembler operating procedures.

Character Set

Each source module consists of a string of ASCII characters. The AOS/VS 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). You can direct the Macroassembler to distinguish between uppercase and lowercase characters by using the /ULC switch on the MASM command line (see Chapter 8).

- Numerals: **0** through **9**

- Question mark: **?**

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

- Special characters:

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

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

Do *not* use any of the following characters in your source module: null (ASCII 000_8), delete (ASCII 177_8), control characters, or characters with the parity bit set to 1. If the Macroassembler encounters one of these, it returns an 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 pseudo-op 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	"Terminators and 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 Operators" - Chapter 2
S(capital S)	Double-precision bit alignment operator (32 bits)	"Bit Alignment Operators" - Chapter 2
~(tilde)	Unary NOT operator (i.e., complement)	"Urinary Operators" - Chapter 2
&(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 and can be changed with .ESCHAR pseudo-op.	"Symbol Names" - Chapter 2
\(backslash)	Generates numbers and symbols	"Generated Numbers and Symbols" - Chapter 5

(continues)

Table 2-1 Special Characters

Character	Meaning	Reference
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)	No-listing indicator; suppresses 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
$(dollar sign)	Expands to a 3-digit number inside macros if the /$ switch is set	"Generated Numbers and Symbols" - Chapter 5

(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

↵

to represent statement terminators.

Examples of source statements are

```
        325 ↵
        XWLDA       5,LOCX ↵
BEGIN:  .ZREL                       ;LOWER PAGE ZERO RELOCATABLE ↵
```

A source statement can not be more that 132 characters in length. If a statement is more that 132 characters long, the Macroassembler will truncate the line and return 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.

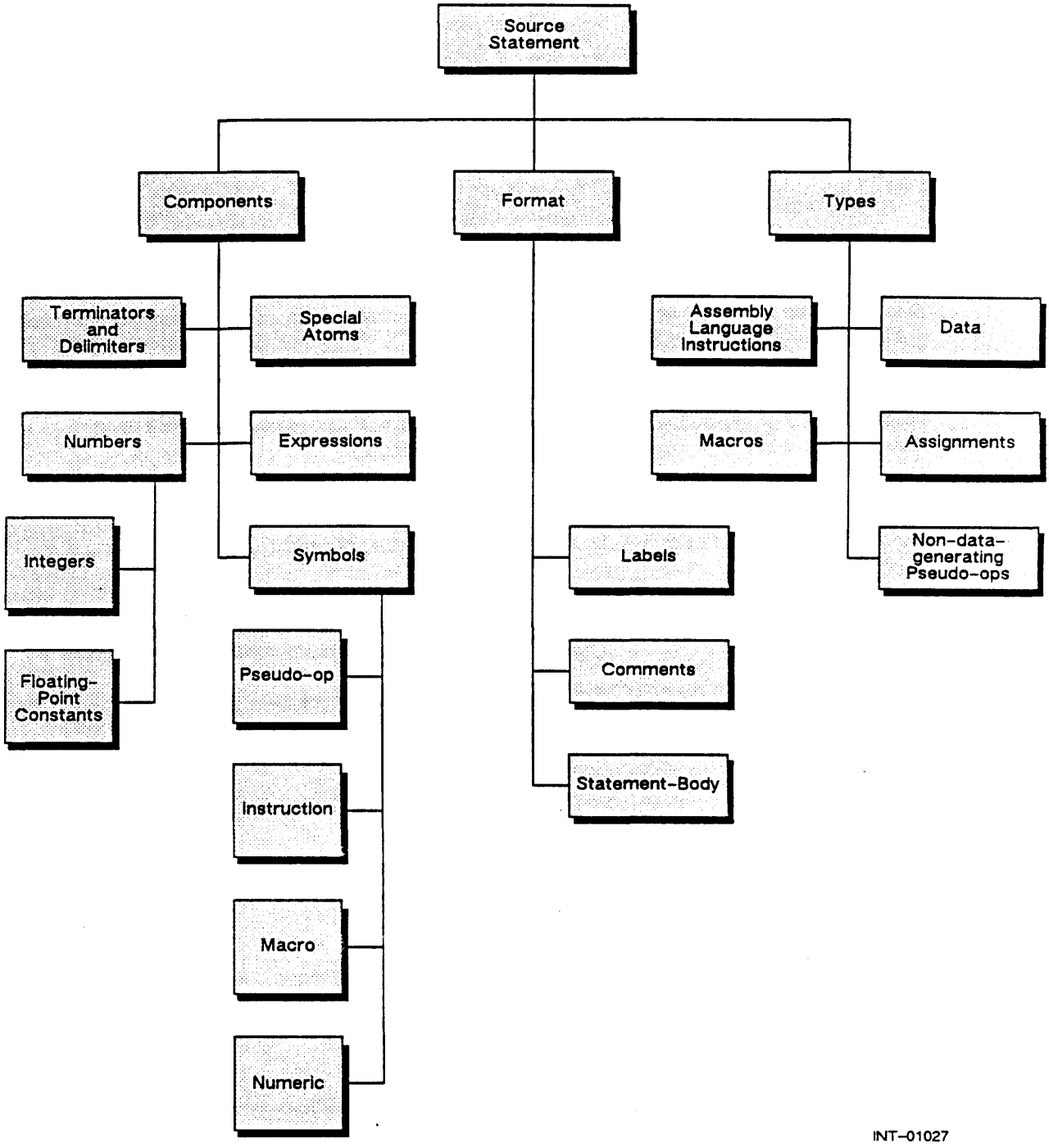


Figure 2-1 The Source Statement

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:

- terminators 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 can 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 should include expressions along with the four types of atoms. Our list of basic statement components is now

- terminators and delimiters
- numbers
- symbols
- expressions
- special atoms

In the following sections of this manual, we discuss 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 the curved down arrow ↵

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)	This symbol represents 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 S	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 can 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 allows you to 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)

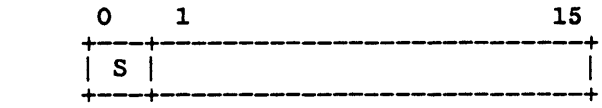
The Macroassembler interprets all integers as double precision (32 bits), by default. You can alter this default storage mode by using the global data placement pseudo-op .ENABLE or one of the local data placement pseudo-ops (.DWORD, .SWORD, .UWORD, or .WORD). Refer to the pseudo-op descriptions in Chapter 7 and also to "Data Placement" in Chapter 6 for more information.

Single and double precision integers can appear in expressions and data statements. Single precision floating-point constants can only appear in data statements; not in expressions.

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 two's 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; must be in the range 0 through 9.
d...	are optional digits in the range of the current input radix.
.	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 can 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 can set the input radix to any base from 8 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
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
0E333
45B
0B2F
```

You can 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,

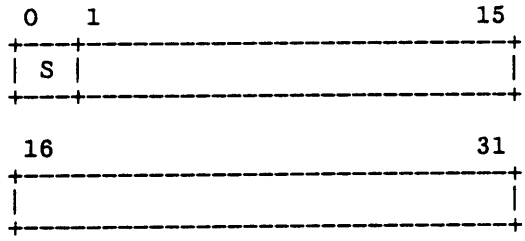
```
.RDX        16          ;Input radix equals 16.
 49B3                   ;The Macroassembler interprets B as
                        ;a hexadecimal digit.
(49)B3                  ;The Macroassembler interprets B as
                        ;the bit alignment operator.
```

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 two's 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*...><.>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; must be in the range 0 through 9.
d...	are optional digits in the range of the current input radix.
	is an optional decimal point. The Macroassembler interprets the integer in base 10 if you supply the decimal point.
break	terminates the integer. The **break** character can be any delimiter or terminator (see "Terminators and Delimiters" earlier in this chapter).

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

```
25      1320    -1      +241            -177777
```

Note that the input format for double precision integers is the same as for single precision integers. Again, this format generates a double precision integer if the Macroassembler is in 32-bit data placement mode (the default mode).

All information presented in the single precision integer discussion also applies to double precision integers. Thus,

- If a decimal point precedes the **break** character, the Macroassembler interprets that integer as decimal.
- The first digit in each integer must be in the range 0 through 9. If the input radix is 11 or greater, your integer can 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).

Refer to the previous section of this chapter ("Single Precision Integers") for more information on these three points.

For compatibility with other Data General assemblers, an integer can be forced to double precision, regardless of the data placement mode, by appending a "D" to it.

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

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

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 (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.

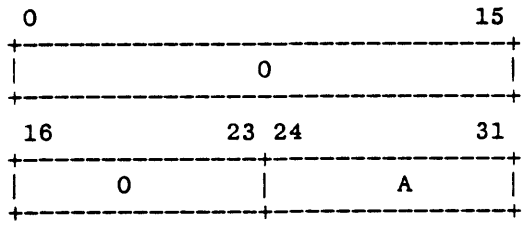
A few examples follow to illustrate the use of the quotation mark.

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

You can 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

The Macroassembler packs the value generated by this format in the rightmost byte of the second word in memory (i.e., in the least significant 8 bits). For example, the Macroassembler stores "A as follows:



If you are in 16-bit data placement mode, the Macroassembler stores only a single-precision integer.

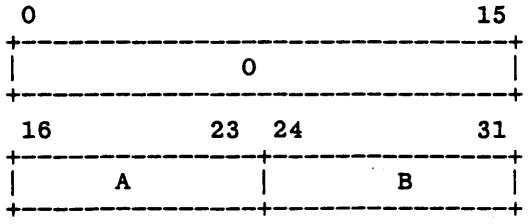
The second special integer-generating format converts up to two ASCII characters into an integer. The format is

'string'

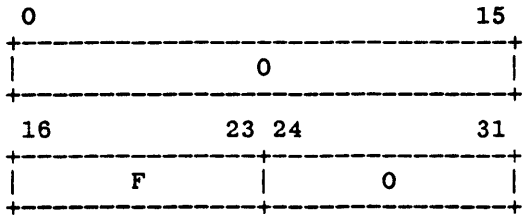
where:

'	is an apostrophe; MASM requires you to enclose the ASCII characters in apostrophies
string	consists of any number of ASCII characters; the Macroassembler uses only the first two characters in this string

The Macroassembler packs the octal values of string's first two characters from left to right in the second word (bits 16–31) of the integer. For example, the Macroassembler stores both 'AB' and 'ABCD' as



If you supply only one character, the Macroassembler places the corresponding octal value in the left byte of the second word. Thus, the Macroassembler stores 'F' as



Two apostrophes without an intervening character string generate an integer containing all zeros (i.e., absolute zero).

If you are in 16-bit data placement mode, the Macroassembler stores only a single-precision integer.

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

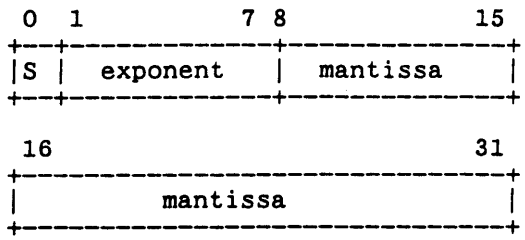
Source	Octal Value
"A	101
'AB'	40502
'BA'	41101
"	0
"+5-2	3
'B'+5	41005
'A'	20101
"A+'A'	40501
'AA'	40501
'AABCD'	40501

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 and in special floating-point instructions (see the *ECLIPSE® MV/Family 32-Bit Principles of Operation* manual).

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

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

You can obtain the negative form of a floating-point number by using the unary negate operator. This has the effect of 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

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

which is approximately

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

The Macrossembler 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 { **E** / **.** } **d break**

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).
d...	are optional digits in the range 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 can be any delimiter or terminal (typically □ ; or ⤵).

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

Floating-Point Constant	Assembled Value	
254.33	141376	052172
254.33E0	141376	052172
25433E-02	141376	052172
25433E-2	141376	052172
2543.3E-1	141376	052172
0.25433E03	141376	052172

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	041766
-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., XWLDA), pseudo-ops (e.g., .NREL), macros, 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 can be any upper- or lowercase letter (A - Z, a - z), period (.), dollar sign ($), or question mark (?)
b	represents succeeding characters in the symbol and can include uppercase and lowercase letters (A - Z, a - z), numbers (0 - 9), period (.), dollar sign ($), question mark (?), and underscore (_)
break	terminates the symbol; a **break** character can 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., %).

```
12.3            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'.

You can direct the Macroassembler to distinguish between uppercase and lowercase letters by using the /ULC switch on the MASM command line. We define all pseudo-ops, MV/Family 32-bit instruction mnemonics, system calls, and system parameters in uppercase letters. Thus, if you use the /ULC switch, be sure to enter these symbols in uppercase.

Also by default, MASM recognizes the first eight 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 8):

```
ASSEMBLY1           ASSEMBLY2           ASSEMBLY_PROGRAM
```

To override the symbol length default value of 8, use the /SYMBOL= switch when building your permanent symbol file. You can use this switch to set the symbol length to any number of characters from 5 to 32 (as described in Table 8-1 and under "Symbol Length" in Chapter 8).

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.

If the /$ command-line switch is set, the meaning of the "$" character is changed so that it is no longer a valid symbol name character. Instead, it is replaced with a three-digit character string wherever it occurs. A unique number is assigned to each macro call, and this character string is a representation of that number. This number may be used for generating unique symbol names inside of macros. See the section, "Generated Numbers and Symbols" in Chapter 5.

Symbol Types

The Macroassembler recognizes four classes of symbols:

- numeric symbols
- instruction symbols
- macro symbols
- pseudo-op symbols

Numeric Symbols

Numeric symbols have a specific numeric value. Because of this value, numeric symbols can be used in arithmetic expressions. A label is an example of a numeric symbol. The numeric value of a label is the address with which it is associated.

Note the following example:

```
VALUE = 3 ↵

START:      XWLDA 0, VALUE ↵
```

The first statement defines VALUE as a numeric symbol. VALUE is a variable with the value 3.

The second statement defines START as a numeric symbol. START is a label whose value is the current address.

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

Numeric symbols can 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 can use it in separately assembled modules. "Inter-module Communication" in Chapter 6 provides more information on this subject.

When choosing numeric 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 numeric symbols do not conflict with any pseudo-op, instruction, or macro symbols. Though MASM permits it, we recommend that you not use the question mark (?) as the first character in your numeric symbols; many system parameters provided by AOS/VS begin with a question mark.

Redefining Numeric Symbols

Generally, you can change the value of a numeric 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, numeric symbol A has the value 0.

Remember that, contrary to the general rule, you can *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.
```

Instruction Symbols

Instruction symbols include all the symbols in the MV/Family 32-bit Assembly Language Instruction Set. The Macroassembler software defines these symbols in an internal permanent symbol table.

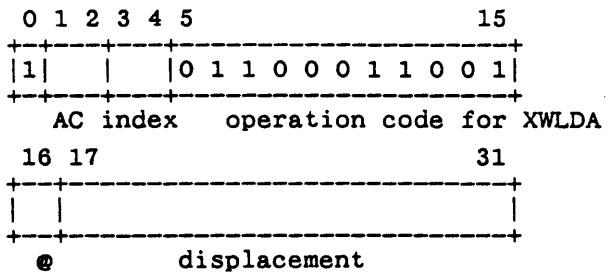
Like numeric symbols, instruction symbols also have an associated numeric value. Unlike numeric symbols, however, instruction symbols also are associated with extra information on the instruction's syntax. This information includes how many words of memory the instruction occupies, how many arguments the instruction requires, and other facts pertinent to the instruction's use.

Each instruction has a specific syntax. As an example, consider the assembly language instruction XWLDA, which loads a value from memory into an accumulator. When using XWLDA, you must specify an accumulator, a displacement value of 15 bits or less, and, optionally, an addressing index. In addition, you may select an indirect addressing mode by including the *at sign* character (@) in the source line's address field. The general syntax for XWLDA is

XWLDA□AC□<@>displacement<□*index*>

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 value of the symbol and its arguments.

For example, when the Macroassembler encounters the instruction symbol XWLDA, it produces a 2-word instruction as follows:



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:

```
FIVE:     5                       ;Location FIVE contains the
          .                       ;value 5.
          .
          .
          XWLDA       6,FIVE      ;ERROR: This source line will cause an
                                  ;error because the value 6 does not
                                  ;fit in the 2-bit field designated
                                  ;for the first argument to XWLDA (the
                                  ;accumulator).
          XWLDA       1,FIVE      ;NO ERROR: The value 1 does fit into the
                                  ;field allocated for the accumulator.
```

As a general rule, be sure that the absolute value of your argument conforms to the following equation:

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

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 syntax	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 *ECLIPSE® MV/Family 32-Bit Principles of Operation* manual describes both of these formats for every assembly language instruction symbol.

Macro Symbols

Macro symbols are associated with lines of text rather than with numeric values. These symbols cannot, therefore, be used in arithmetic expressions. Like instruction symbols, macro symbols are associated with formats and can have arguments.

Every macro symbol represents a series of assembly language source statements. Whenever you want 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.

Defining Macro Symbols

Use the .MACRO pseudo-op to associate a series of source statements with a symbol. You can choose any unique symbol name to represent those source statements.

The format for using .MACRO is

```
.MACRO□macro-symbol
source-statements
    .
    .
    .
%
```

We provide a complete discussion of macros in Chapter 5. Also, refer to the .MACRO pseudo-op description in Chapter 7 for more information.

Redefining Macro Symbols

You use .XPNG to remove any macro symbol's definition (including system call definitions). After deleting a macro symbol's definition, you can 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.

Pseudo-Op Symbols

Pseudo-op symbols reside within the Macroassembler software (in the permanent symbol table) and define all the pseudo-ops. The 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*. Pseudo-ops that represent internal assembler variables are called *value symbols*. Some pseudo-ops may be used as either assembler directives or value symbols.

You can use the .LOC pseudo-op, for example, as an assembler directive in the following source code:

```
.LOC                    100
.DWORD                  5
```

This code uses the pseudo-op symbol .LOC to set the value of the location counter to 100. The Macroassembler then stores the number 5 in two memory words starting at absolute location 100.

The following source code, on the other hand, illustrates how you can use .LOC as a value symbol:

```
.PUSH                   .LOC
```

This code uses the pseudo-op symbol .LOC to represent the current value of the location counter. The Macroassembler pushes this value onto the stack. Appendix B provides a summary of pseudo-ops, and tells which pseudo-ops can be used as assembler directives and which as value symbols.

Expressions

An *expression* is

- a single numeric symbol, value symbol, or integer, or
- a series of numeric 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	can be a numeric 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 can 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 can not include any spaces within an expression, but you can use a space to terminate an expression. Thus, the Macroassembler does not view the string '3 + 5' as an expression because it contains spaces.

Operators

Table 2-6 lists all the operators that the AOS/VS Macroassembler recognizes.

Table 2-6 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)
	S	Double precision bit alignment (32 bits)
Logical Operators	&	Logical AND
	!	Logical OR
	~	Unary NOT (complement)
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. For example, the following expressions contain binary operators:

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

You *cannot* place two binary operators in a row. The following are *illegal* expressions:

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

All the operators in Table 2-6 may function as binary operators, except for - (the unary NOT operator).

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. In addition to the NOT operator (-), + and - can also function as unary operators. We provide more information in the next section.

Unary Operators

The Macroassembler recognizes the three unary operators listed in Table 2-7.

Table 2-7 Unary Operators

Operator	Operation
+	Unary plus
-	Unary minus
-	Unary NOT (complement)

Unary operators require only one operand, which must appear immediately after the operator. Thus, a unary operator can 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        5*-2

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

Use parentheses to separate the operators when applying two or more consecutive unary operators. The following expressions illustrate this:

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

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

The *unary NOT operator (-)* directs the Macroassembler to complement each bit of the following operand. That is, the result in a given bit position is 1 if the operand contains a 0 in that bit position; the result is 0 if the operand contains a 1 in that bit position.

The unary NOT operation for the expression -3 is shown below.

Bit representation of 3:	00 000 000 000 000 000 000 000 000 000 011
Result of - operation:	11 111 111 111 111 111 111 111 111 111 100

Thus, the value of the expression -3 is 37777777774_8.

Note that the Macroassembler operates on all 32 bits of the operand. In the above example, all leading zeros become ones.

Logical Operators

In addition to the logical NOT operator, 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 not to use relocatable operands in a logical expression; i.e., 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 zeros in all bits except the least significant bit (bit 31) which contains a one. 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	000000	5==6	False
000000	000001	3==3	True
000000	000001	7>1	True
000000	000000	55<=41	False
000000	000001	7<>6	True

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

Bit Alignment Operators

The AOS/VS Macroassembler recognizes two *bit alignment operators:* B and S. These two operators allow you to right justify an integer on a bit boundary.

The general format for using the bit alignment operators is

operand bit-op position

where:

operand	is an integer, symbol, or expression whose value you want to align
bit-op	is a bit alignment operator (B or S)
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.

The S Operator

The *S operator* and the B operator are similar; however, S generates a double precision (32-bit) integer while B generates a single precision (16-bit) integer. The value of the **position** argument must be in the range

$$0 <= \text{position} <= 31_{10}$$

The result of an S bit alignment expression equals

$$\textbf{operand}_r * 2^{(31 - \textbf{position})}$$

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 show the use of S; the radix equals 8:

Assembled Value		Expression	Comment
000000	000001	1S31	Aligns the value 1 in bit position 31_{10}.
000060	000000	3S11	Aligns the rightmost bit of 3_8 (11_2) in bit position 11_{10}.
140000	000000	7S1	Aligns the rightmost bit of 7_8 (111_2) in bit position 1. The rest of 7 is lost.
000000	000000	3S45	ERROR: the value 45 is outside the legal range (there is no bit 45_{10}).

The B Operator

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

$$0 <= \textbf{position} <= 15_{10}$$

The result of a B bit alignment expression equals

$$\textbf{operand}_r * 2^{(15 - \textbf{position})}$$

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

If the Macroassembler is in 32-bit data placement mode, the first 16 bits of the result will contain zeros. In addition, the value of **position** will indicate a bit in the second word of the data representation. For example, a **position** of 0 identifies the first bit of the second word (bit 16 in 32-bit notation).

The following examples illustrate the use of B (radix equals 8; 32-bit data placement mode):

Assembled Value		Expression	Comment
000000	000004	1B13	Aligns the value 1 in bit position 13_{10} of the second word (i.e., $0\text{-}000\text{-}000\text{-}000\text{-}000\text{-}100_2$).
000000	000012	5B14	Aligns the rightmost bit of 5_8 (101_2) in bit position 14_{10} of the second word (i.e., $0\text{-}000\text{-}000\text{-}000\text{-}001\text{-}010_2$).
000000	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).
000000	070000	27B3	Aligns the rightmost bit of 27_8 ($010\ 111_2$) in bit position 3 of the second word. MASM truncates the portion of 27 that does not fit into the second 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.
```

Using Bit Alignment Operators With Symbols

In the previous bit alignment examples, all operands are integers. If you pass symbols to the bit alignment operators, the assembler can misread the B or S as part of the symbol; that is, MASM may not recognize the B or S as an operator. Therefore, if the operand preceding the bit alignment operator is a symbol, enclose that argument in parentheses. The following examples illustrate this:

Assembled Value		Expression	Comment
000000	000025	A=25	Input radix equals 8. Numeric symbol A has the value 25_8.
000000	000006	AS9=6	Numeric symbol AS9 has the value 6.
002500	000000	(A)S9	Aligns the rightmost bit of 25_8 ($010\ 101_2$) in bit 9.
000000	000006	AS9	The Macroassembler generates a storage area with the value of symbol AS9.
100000	000000	(A)S0	Aligns the rightmost bit of 25_8 in bit 0; the rest of 25 is lost.
124000	000000	(A)S4	Aligns the rightmost bit of 25_8 in bit 4.
030000	000000	(AS9)S4	Aligns the rightmost bit of 6_8 (110_2) in bit 4.

When using bit alignment operators in a lengthy expression, enclose both operands in parentheses to ensure that the assembler aligns the value correctly. The following example demonstrates this:

Assembled Value		Expression	Comment
000000	000025	A=25	Radix equals 8.
000000	000010	C=10	
000320	000000	(A-C)S11	(A-C) equals 15_8 so this expression is the same as 15S11.
014000	000000	(3B12)S(3+4)	(3B12) equals 30_8. Thus, the expression equals (30)S(7).

Priority of Operators

You can 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 S	3 (highest priority)
+ - * / & ! -	2
< <= > >= == <>	1 (lowest priority)

If an expression contains operators of equal priority, the assembler evaluates them from left to right. Using parentheses in forming expressions greatly improves clarity.

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

Assembled Value		Expression	Comment
000000	000001	3*2==6	MASM evaluates * first, then ==. The relationship is true.
000000	000000	4<6-3	MASM evaluates - first, then <. The relationship is false.
000000	000005	4/2+3	/ and + are equal in priority so the assembler evaluates them from left to right (first /, then +).
000000	000011	3*2-1+4	All operators are of equal priority so MASM evaluates them from left to right.
000000	000006	3!1*2	Both operators are of equal priority so MASM evaluates them from left to right (3!1=3; (3)*2=6).
000000	000001	4&2<>3!1	MASM evaluates & first followed by ! and lastly <>. The resulting relationship is true.
000000	030000	2*3B4	B has higher priority than *, so the assembler evaluates 3B4 first. It then multiplies the result by 2.

You can 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
000000	000006	2*(4-1)	MASM performs operations in the following order: (4-1)=3 2*(3)=6
000000	000002	1+(6/2)/2	Order of operations: (6/2)=3 1+(3)=4 (4)/2=2
000000	000004	(3*2)-(4/2)	Order of operations: (3*2)=6 (4/2)=2 (6)-(2)=4
000000	000001	(5<=5)+(6==2)	Order of operations: (5<=5)=1 (6==2)=0 (1)+(0)=1
000000	000006	3*((3+1)/2)	Order of operations: (3+1)=4 ((4)/2)=2 3*((2))=6
000000	000025	A=25	Radix equals 8.
000000	000010	C=10	
000000	010025	A+(C*2)S23	(C*2)S23 is equivalent to (20)S23 and equals 10000_8. A+10000_8 equals 10025_8.

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. Other 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 adhere to 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 can include the at sign (@, also called the *indirection indicator*) in certain memory reference instructions (MRIs) or in any data statement. When the Macroassembler encounters the @ character, it sets a specific bit (the indirect addressing bit) to 1.

In memory reference instructions, the @ sign must appear at the beginning of the displacement field. In one-word memory reference instructions, bit 5 is the indirect addressing bit. In longer memory reference instructions, the first bit of the displacement field is the indirect addressing bit. Refer to the *ECLIPSE® MV/Family 32-Bit Principles of Operation* manual to determine which memory reference instructions allow indirect addressing.

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

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

Assembled Value		Source	Statement
024420		LDA	1,20
026420		LDA	1,@20
101431	000020	XWSTA	0,20,0
101431	100020	XWSTA	0,@20,0
000000	000025	25	
100000	000025	@25	

Asterisks (**)

Two consecutive asterisks (**, also called the *no-listing indicator*) at the beginning of a source line direct the Macroassembler not to list that line. The asterisks do not affect the object module, only the assembly listing. The /XPAND switch can be used to override the listing suppression.

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

Number Sign (#)

You use the number sign character (#, also called the *no-load indicator*) in conjunction with ECLIPSE 16-bit arithmetic and logic class (ALC) instructions. These instructions include ADD, SUB, ADC, MOV, AND, COM, and NEG.

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).

When using the # character, place it immediately to the right of the assembly language instruction symbol.

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

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

Do not use the # character in a source line in combination with either the always skip (SKP) or never skip (no mnemonic) options. If you do, MASM returns an error because those bit combinations represent other instructions in the MV/Family 32-Bit instruction set.

Refer to the *ECLIPSE® MV/Family 32-Bit Principles of Operation* manual for a list of the skip mnemonics and for information about ALC instructions.

Statement Format

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

```
label:          statement-body          ;comment ↵
```

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

The AOS/VS Macroassembler is a free-form assembler. That is, the assembler is not column senstitive but 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 can 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 ↵

        label:      statement-body      ;comment ↵

label:              statement-body      ;comment ↵
```

You can 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 all lines that are too long.

If you include the two asterisks (**) no-listing indicator on a source line, you should make sure that no characters precede the asterisks:

```
        ** label:   statement-body      ;comment ↵
```

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 show how to use labels:

```
BEGIN:    XWLDA     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 exceptional case at the end of this section.) You can 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 XWLDA 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. This statement 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 can appear alone on a line, in which 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 can 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:          LWLDA       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 LWLDA 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
.PART
.ZREL
```

In general, you should not place labels on these pseudo-op statements. In fact, except for .BLK, .TXT, and the local data placement pseudo-ops (.DWORD, .SWORD, .UWORD, and .WORD), 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 can include *comments* in your program to facilitate program development, maintainance, and documentation. The assembler does not interpret comments and, thus, 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:

```
        LWLDA       0,A         ;Load AC0 with the contents of location A
        LWLDA       1,B         ;Load AC1 with the contents of location B
        WADD        0,1         ;Add AC0 and AC1 together

A:      63                      ;Value for AC0
B:      44                      ;Value for AC1

        ;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 intructions
- 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

label: { **assembly language instruction** / **macro or system call** / **pseudo-op directive** / **assignment** / **data** } **;comment** ↵

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

The following sections of this manual describe the five statement types and the syntax for each.

Assembly Language Instructions

Assembly language instructions perform specific operations at execution time. All assembly language instructions fall into the following three categories:

- input/output instructions (I/O)
- memory reference instructions (MRI)
- arithmetic and logic class instructions (ALC)

Use *I/O instructions* to 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
- test the status of a device

Note that you can *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" below).

Memory reference instructions (MRI) allow you to 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

Arithmetic and logic class (ALC) instructions allow you to 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).

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 *ECLIPSE® MV/Family 32-Bit Principles of Operation* manual describes the MV/Family 32-bit assembly language instructions and specifies what arguments you must supply to each. Examples of MV/Family 32-bit assembly language instructions are

```
XWLDA     0,-2,1
ADD       2,3
XJMP      START
HALT
```

Macros

A *macro* is a series of assembly language source statements which you assign a name. Whenever you want 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.

Pseudo-Ops

A *pseudo-op*, also called an *assembler directive*, directs the operation of the Macroassembler. It is called a "pseudoinstruction" 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
- define macros

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 "Pseudo-Op 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 the "Assignments" section).

When using a pseudo-op symbol in this fashion, we refer to it as a *value symbol*. We discuss value symbols under "Pseudo-Op 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 double precision (32-bit) integer value to a symbolic name. After associating a value with a symbol, you can use the symbol any time you want to indicate the value.

The syntax of an assignment statement is

numeric-symbol = { **integer** | **symbol** | **expression** | **instruction** }

where:

numeric-symbol	is a numeric symbol conforming to the rules for symbols (given earlier in this chapter); the Macroassembler assigns **numeric-symbol** the value on the right side of the = character
integer	is any integer value; you can *not* place a floating-point number on the right side of an assignment statement
symbol	is any numeric symbol or value symbol
expression	is any legal expression
instruction	is any legal MV/Family 32-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=XWLDA    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. You should avoid including instructions within expressions. If you must include an instruction within an expression, enclose the instruction in parentheses.

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
```

You should use data placement pseudo-ops to control whether MASM generates single or double-word storage areas. Use the .DWORD pseudo-op to store data in two words of memory, and use the .WORD pseudo-op to store data in one word of memory. The .WORD pseudo-op truncates the high-order word of data before storing the result in 16 bits of memory. The use of these pseudo-ops is illustrated in the following assembly language program:

```
  Location  Data              Source
  Counter   Field             Code
-------------------------------------
01                            .TITLE TEST
02          00000000001       .NREL 1
03 000000 SC 000003           .WORD 3
04 000001 SC 00000000004      .DWORD 4
05 000003 SC 000005           .WORD 5
06                            .END
```

Note that the numbers 3 and 5 are stored in one data word each by using the .WORD pseudo-op. By contrast, the number 4 is stored in two data words because of the .DWORD pseudo-op. Notice that the location counter jumps from 000001 in line 04 to 000003 in line 05, indicating that the number 4 takes up two words of memory storage (see "Data Placement" in Chapter 6 for more information).

Other useful data placement pseudo-ops include .SWORD and .UWORD. The .SWORD and .UWORD pseudo-ops act the same as .WORD but provide error messages when truncation from 32 bits to 16 bits results in an error (see the "Pseudo-Op Dictionary" in Chapter 7, and "Data Placement" in Chapter 6 for more information).

The .TXT pseudo-op allows you to store a string of ASCII characters in memory. For example the following source code:

```
        .TXT %hello%
```

when assembled produces three words of data storage as shown below:

```
  Location  Data              Source
  Counter   Field             Code
-------------------------------------
03 000000 SC 064145 066154    .TXT %hello%
04           067400
```

These three data words contain the ASCII code for the lowercase letters "hello" packed two letters per word. The last word, 067400, contains the ASCII code for "o" in its most-significant byte, and all zeroes in its least-significant byte.

You can 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 must immediately precede the data on the source line; for example:

```
@113
@1032
```

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

End of 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*.

To produce an object module, the Macroassembler must scan your source code twice; that is, MASM is a *two-pass assembler*. On the first pass, MASM finds the names and values of all the symbols and keeps a tally of how many words of code and data there are. On the second pass, MASM resolves the final value of all the words of code and data, and puts them in the object file.

MASM needs to make two passes in order to allow your source program to make use of *forward references*. A forward reference occurs whenever an assembly language statement includes the name of a memory location that is situated further on in the code. For example, if your program contains the statement:

```
XWLDA 0,NEXTPTR
```

and further down in the code contains:

```
NEXTPTR:  .DWORD -1
```

then MASM could not assemble the first statement on only one pass. MASM cannot assign NEXTPTR a numerical value until MASM encounters the second statement. It is only then that MASM assigns NEXTPTR the value of its location within the program. MASM must then go back and plug in NEXTPTR's value in the first statement.

During the 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 AOS/VS 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 four types of symbols that can appear in your source module (i.e., numeric, instruction, macro, and pseudo-op). As the Macroassembler processes your source module, it translates all symbols into their binary equivalents.

The Macroassembler uses two different tables to resolve the symbols in your source code:

- permanent symbol table
- temporary symbol table

During the assembly process, the Macroassembler makes sure that all symbols are defined in one of these tables (adding definitions from your source, if necessary). On its second pass through your program, the assembler uses these tables to substitute binary code for the symbols in your source module.

The following discussion focuses on the creation and use of these two symbol tables. The flowchart in Figure 3-1 parallels this discussion and can help you understand the symbol resolution process.

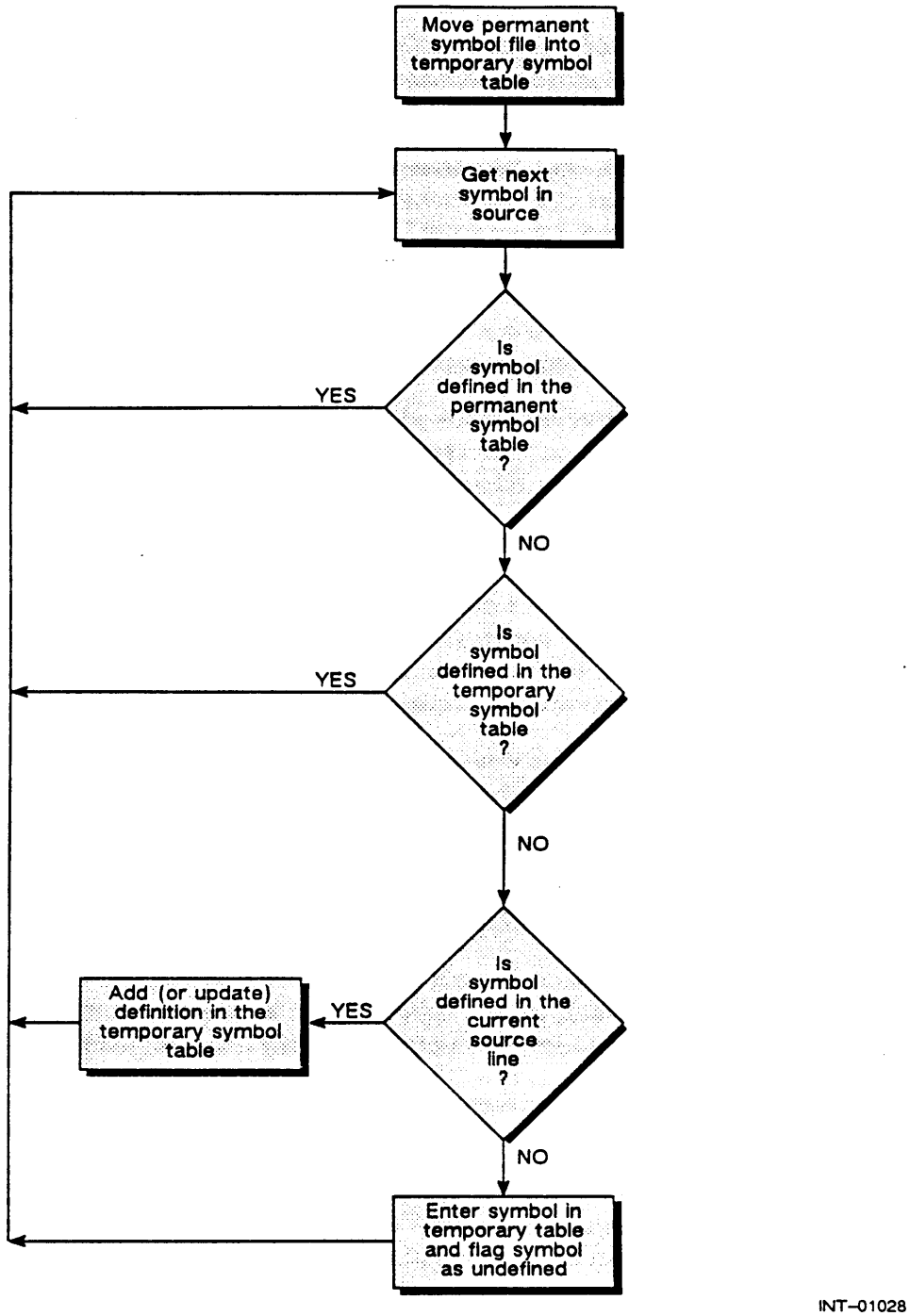


Figure 3-1 Resolving Symbols

Symbol Tables

The Macroassembler software defines all pseudo-ops and all instructions in the ECLIPSE MV/Family instruction set in the *permanent symbol table*. Thus, from the outset, the MASM can recognize all pseudo-op and instruction symbols in your source module.

Upon executing MASM, the Macroassembler automatically creates a *temporary symbol table*. Initially, this table does not contain any information. However, when you define a symbol (such as a label, a macro, etc.) in your source code, the symbol is put into the temporary symbol table unless there is already a symbol with the same name in either the permanent or the temporary table. Thus, by the end of the assembly process, this table will contain a definition for each symbol in your program, except pseudo-ops and MV/Family 32-bit instruction mnemonics.

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 the symbol is not legal.

If the symbol is valid, the Macroassembler looks for its definition in the permanent symbol table. When looking up symbols in the permanent symbol table, MASM uses only the first eight characters of the name. In addition, MASM accepts all pseudo-op names in either upper- or lowercase. If MASM finds the symbol in the permanent table (i.e., if the symbol is a pseudo-op or instruction), the assembler does not have to check any other tables for that symbol's definition.

If the permanent symbol table does not contain a definition for the symbol, the Macroassembler checks the temporary symbol table. When looking up or adding symbols to the temporary symbol table, MASM uses only the first eight characters of the name. All names are case-insensitive, by default. You can change to case-sensitive by using the /ULC switch, and you can change the 8-character recognition limit to a number between 5 and 32 by using the /SYMBOLS=<n> switch.

If the Macroassembler does not find a symbol's definition in either of the tables or in the 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. This is because of forward references (see the beginning of this chapter).

If, during pass two, the Macroassembler encounters a symbol that is not defined in either 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 inter-module communication pseudo-op), the Macroassembler does not return an error. (Chapter 6 contains more on inter-module communication.)

Permanent Symbol Files

To speed up assembly, you can "pre-assemble" any number of assignment statements and macros, save the resulting temporary table in a file, and reuse it again and again. This type of file is called a permanent symbol, or .PS file. For example, suppose you have written a short program (here called MYPROG.SR) that makes several system calls. Normally, you would need to include all the AOS/VS system call parameter files in your assembly:

```
XEQ MASM/0=MYPROG.OB PARU.32.PR SYSID.32.SR MYPROG.SR
```

Even if your program is quite short, this assembly will take a long time because of the length of PARU.32.SR and SYSID.32.SR. You can get around this delay by "pre-assembling" your own .PS file (here called MYMASM.PS) which contains both PARU.32.SR and SYSID.32.SR:

```
XEQ MASM/MAKEPS/PS=MYMASM.PS PARU.32.PR SYSID.32.SR
```

Having done this, you can now assemble MYPROG much more quickly by using your MYMASM.PS file instead of PARU.32.SR and SYSID.32.SR:

```
XEQ MASM/PS=MYMASM.PS MYPROG.SR
```

If you do not create your own .PS file, MASM uses file MASM.PS as the default permanent symbol file. MASM.PS contains definitions for all AOS/VS system calls. MASM moves the permanent symbol file into the temporary symbol table at the start of the assembly process. This allows MASM to recognize all AOS/VS system calls in your source module from the outset.

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 performs this statement evaluation process. Figure 3-2 provides an overview of the operations involved and will help you understand the following discussion.

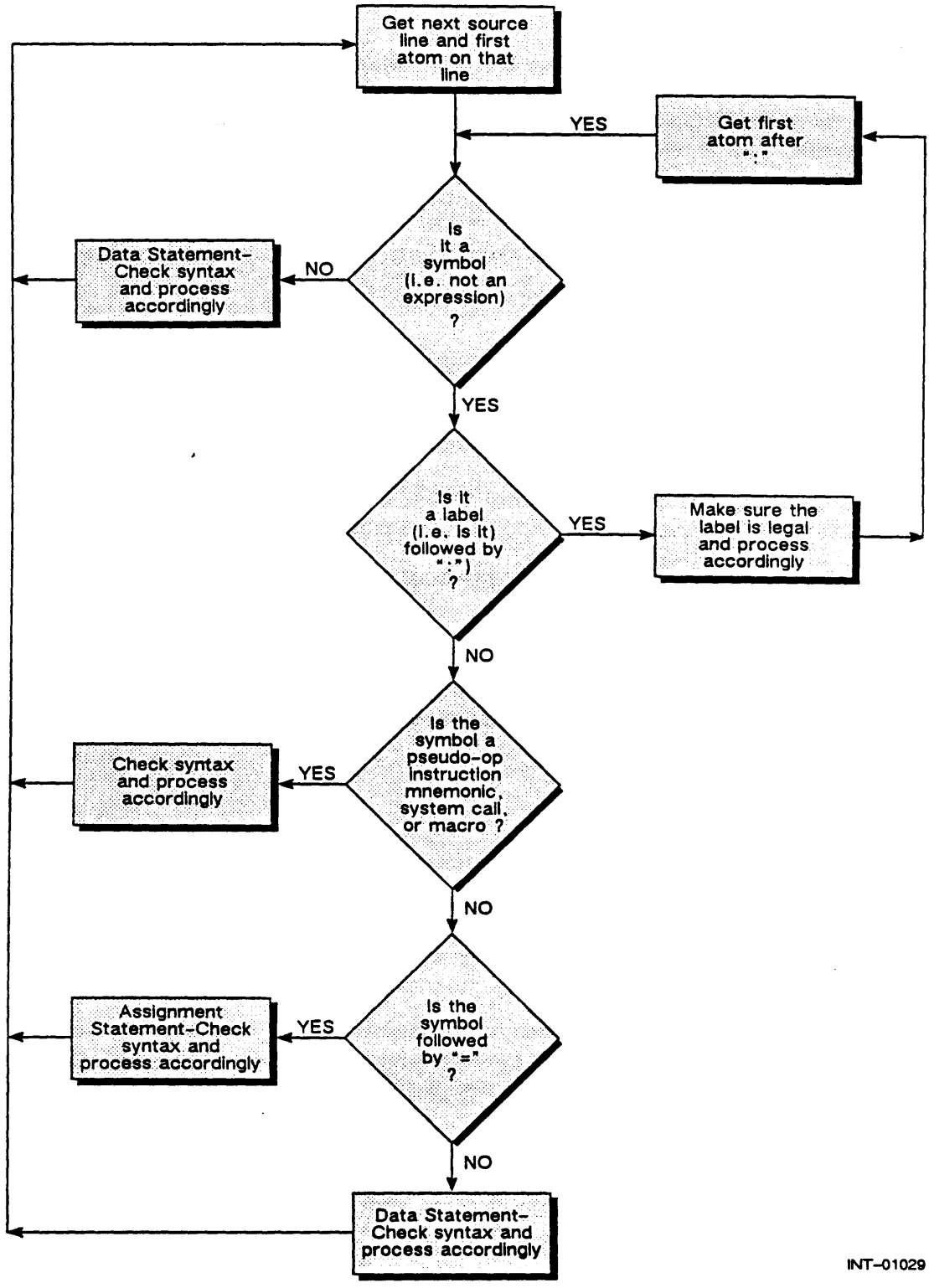


Figure 3-2 Processing Source Statements

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, on the other hand, the 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 can 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.

If the first atom on the source line is a symbol and is not followed by a colon, MASM checks the symbol tables to see if it is a pseudo-op, an instruction mnemonic, 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:

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

If the first symbol on a source line is a pseudo-op, instruction mnemonic 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 — Expands it (see "Processing Macros" later in this chapter)

If the first atom on the source line is a symbol but is not a label, pseudo-op, instruction, 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, 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

Chapter 2 describes macros. To review, a macro is a section of source code that you assign a name. Whenever you want to insert that code in your module, simply specify the macro name. AOS/VS system calls are pre-defined as macros.

The following discussion explains how the Macroassembler processes macros. Refer to Chapter 5 for information about using macros in your source module.

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 onto 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

When it encounters a macro call, MASM searches the permanent and temporary symbol tables for that entry. All macro definitions you supply in your source module reside in the temporary table.

If MASM does not find the macro symbol in either the permanent or temporary symbol tables, MASM then searches your permanent symbol (.PS) file.

When it has found the macro definition, MASM expands the macro. That is, the Macroassembler processes the macro definition string that resides in the temporary symbol table as if it were 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 can control this process.

Memory

In order to explain how assembly language programs are stored in memory, it is necessary to define the concept of *memory partitions*. A partition is a contiguous section of memory that possesses one common set of attributes.

These attributes determine whether the information in the partition is data or executable code, is shared or unshared, global or local. In addition, the attributes determine how the information is aligned within the partition, and whether the information can be overwritten without generating an error message (see the "Partition Attributes" section of this chapter for more detail on attributes).

In general, you create and define memory partitions using the .PART pseudo-op. However, there are also several predefined partitions that you can access by using the .ZREL and .NREL pseudo-ops.

These three pseudo-ops, .PART, .ZREL, and .NREL, are not executed during assembly time. Instead, they are interpreted as orders which the assembler passes to the linker. The Link utility, in turn, sorts sections of your program and stores them in the proper partitions. Figure 3-3 shows sections of source code sorted into different partitions.

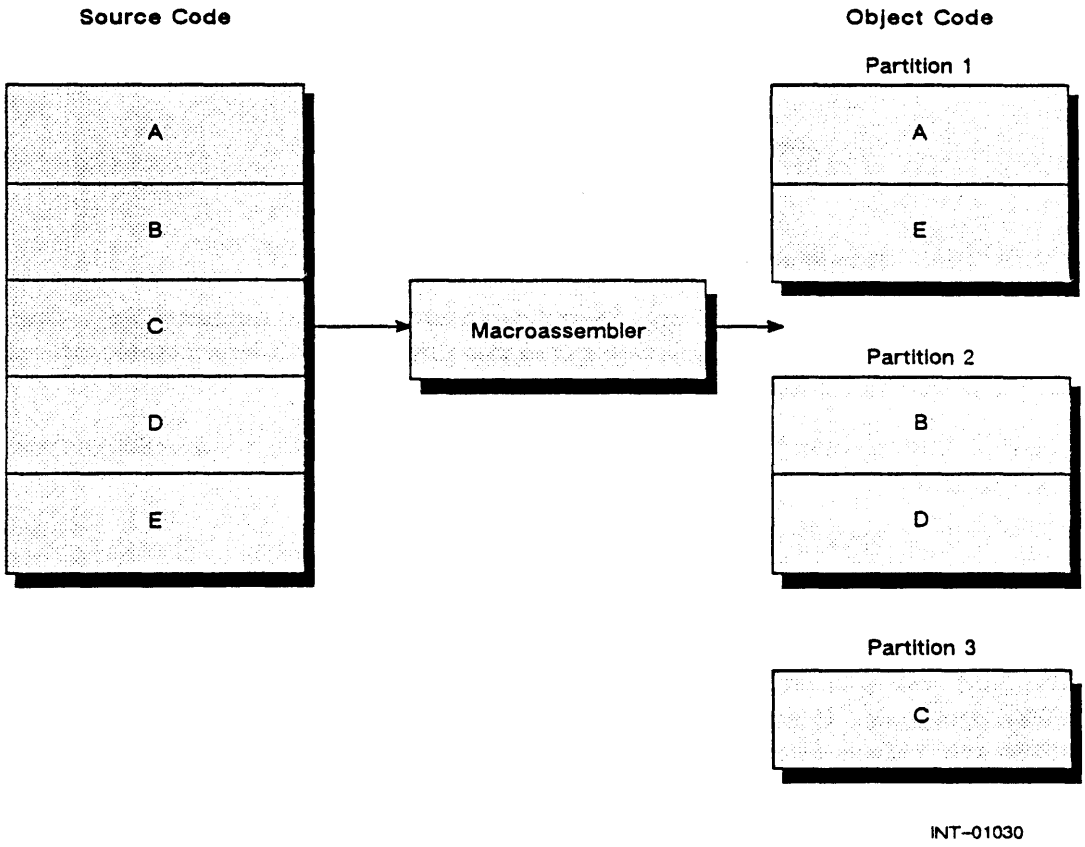


Figure 3-3 Sorting Code into Memory Partitions

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

Logical Address Space

Each process running under AOS/VS has access to 2^{31} words of virtual memory called the *logical address space*. The lowest address in the logical address space is 0, and the highest address is $2^{28} - 1$, the largest number that can be represented with 28 bits.

This logical address space can be subdivided as shown in Figure 3–4, into three areas. The first forty locations, 0–39, are reserved (see *ECLIPSE® MV/Family 32-Bit Principles Of Operation* for more information). The topmost area, called ZREL, extends from addresses 40 to 255. Any code stored in this area can be accessed with only 8 bits of address.

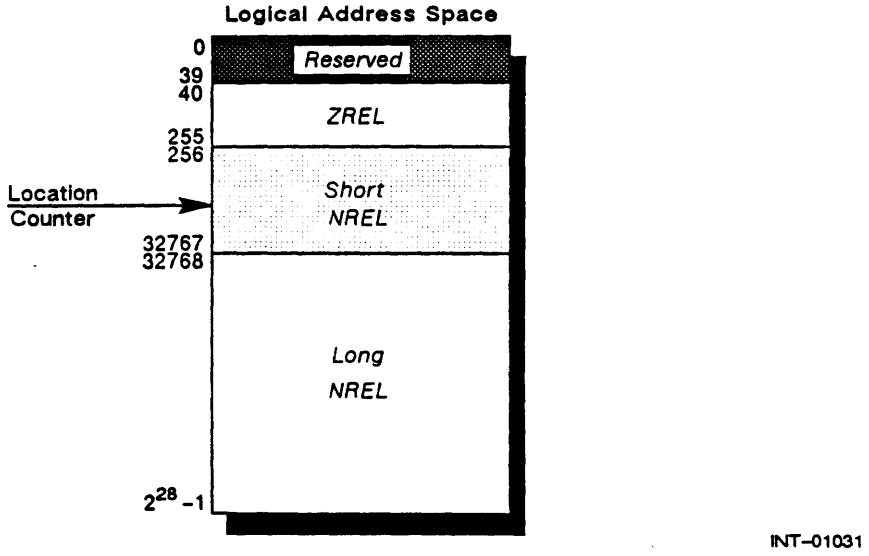


Figure 3–4 Organization of Logical Address Space

The second area begins at address 256 and can extend as far as address 32767. This area is called Short NREL. Any data stored in Short NREL can be accessed with only 15 bits of address, since $2^{15} - 1$ equals 32767.

Use the .NREL pseudo-op with an argument of 2 to indicate this predefined partition. For example:

```
.NREL  2
.              ;These words reside
.              ;in the predefined
.              ;Short NREL partition.
```

The third area is called Long NREL. Link places Long NREL code wherever Short NREL data ends. If Short NREL were entirely full, Long NREL would start at location 32768 and perhaps extend out to the end of the logical address space. Code stored in Long NREL requires full 31-bit addresses. (The logical address space is divided into eight segments. Three of the 31 address bits determine the current segment of execution. See the *ECLIPSE® MV/Family 32-Bit Principles Of Operation* for more information on segments.)

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 later in this chapter) of the next memory location that MASM will assign. In other words, during assembly, the location counter points to MASM's position in the logical address space.

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

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).

Relocatable and Absolute Code

All code placed using the .ZREL, .NREL, and .PART pseudo-ops is *relocatable*. This means that the code does not need to be placed at any specific location in memory; anywhere within the partition will do. Most of the program code you write will be relocatable code.

Although the position within memory of a block of relocatable code may vary, the relationships between elements within the block of code remain constant. That is, a block of relocatable code is relocated as a block; the relative positions of instructions within the block are maintained.

Occasionally, you may need to specify precisely where within the logical address space a section of code will reside. For example, you may need to ensure that an error-handling routine starts at a specific address. To do this, you use the .LOC pseudo-op. Code that you place using the .LOC pseudo-op is *absolute* code, provided the argument used with .LOC is an absolute expression (see "Absolute Expressions" in this chapter).

For example:

```
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).

In general, you do not store your source code in absolute locations. Instead, you use the relocatable partitions (described in 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 can redefine ZREL to extend from locations 100_8 to 377_8 instead of from 50_8 to 377_8. Then, you can use locations 50_8 to 77_8 as absolute addresses and ZREL code will not overwrite them. Refer to the *AOS/VS Link and Library File Editor (LFE) User's Manual* for information about how to change relocatable partition boundaries.

Partition Attributes

There are six different types of partition attributes:

- shared or unshared
- code or data
- normal base or common base
- alignment
- overwrite-with-message or overwrite-without-message
- global or local (user-defined partitions only)

We discuss these attributes in the following sections of this manual. The *AOS/VS Link and Library File Editor (LFE) User's Manual* also describes the various partition attributes.

Shared and Unshared Attributes

The *shared* and *unshared* attributes determine whether several processes can execute the code in a partition at the same time. Shared partitions, which are usually write protected, must reside in the NREL portion of memory. Unshared partitions can reside in either ZREL or NREL memory.

Refer to the *AOS/VS Link and Library File Editor (LFE) User's Manual* for more information about shared and unshared memory partitions.

Code and Data Attributes

These attributes determine whether your partition contains code (executable instructions) or data (constants, variables, text, etc.). In general, use *data* partitions for memory words that your program will not execute at runtime. *Code* partitions can contain both executable and nonexecutable words of memory.

Normal and Common Base Attributes

The normal and common base attributes determine how Link relocates similar partitions from separately assembled modules. Link arranges *normal base* partitions in consecutive order on a module by module basis. Thus, separately assembled partitions with similar attributes follow one another in the program file. If separately assembled partitions have a *common base,* Link does *not* arrange them in consecutive order but uses the same relocation base for each one. That is, Link relocates each contribution as an offset from the same memory location.

Alignment Attribute

The *alignment* attribute directs Link to align the contents of a partition on a power-of-two word boundary. Partitions may begin on any power-of-two word boundary (2^n word boundary) from a single word boundary (2^0) to a 1K-word boundary (2^{10}). A partition with an alignment factor of 2^0 can begin on any word boundary; that is, the partition is *word aligned.*

Overwrite-with-message and Overwrite-without-message Attributes

These attributes control whether Link returns a warning message if it overwrites some code in a partition. If a partition has the *overwrite-with-message* attribute, Link issues a warning when it places data in a location that already contains a nonzero value. The *overwrite-without-message* attribute directs Link to suppress these warning messages.

Global and Local Attributes

The global and local attributes apply only to user-defined partitions (that is, partitions you define with the .PART pseudo-op). Suppose you have written two different modules, each with a user-defined partition as shown below.

Module A

```
.PART MINE  LONG,GLOBAL,MESS
            .
            .
            .
.END
```

Module B

```
.PART MINE  LONG,GLOBAL,MESS
            .
            .
            .
.END
```

The partitions in each module are named 'MINE' but contain different code. If you assign each partition the global attribute, Link will consider the two partitions to be one and the same. Code from module A will be placed in partition 'MINE' along with code from module B.

If you assign either or both of the partitions the local attribute, Link will create separate 'MINE' partitions for modules A and B. Code from each module will then be segregated.

Partition Types

Each memory partition has one attribute from each of the categories described previously. The Macroassembler recognizes two types of partitions:

- predefined partitions
- user-defined partitions

MASM defines certain partitions for your use; we refer to these as *predefined* partitions. They represent the combinations of attributes that you usually use in your program. Table 3-1 lists the predefined partitions and their attributes.

Table 3-1 Predefined Memory Partitions

Partition Name	Attributes	Pseudo-Op
ZREL	ZREL unshared normal base word alignment code overwrite-with-message	.ZREL
Short NREL	NREL (short) unshared normal base 1-word alignment data overwrite-with-message	.NREL 2
Unshared NREL code	NREL (long) unshared normal base word alignment code overwrite-with-message	.NREL, .NREL 0, or .NREL 4
Shared NREL code	NREL (long) shared normal base word alignment code overwrite-with-message	.NREL 1, or .NREL 7
Unshared NREL data	NREL (long) unshared normal base word alignment data overwrite-with-message	.NREL 6
Shared NREL data	NREL (long) shared normal base word alignment data overwrite-with-message	.NREL 5

To place data in a predefined partition, you must issue the appropriate pseudo-op. The Macroassembler creates a partition only if you direct code to that partition. Therefore, each object module does not contain every predefined partition type; it contains only the necessary ones.

A *user-defined* partition is one that you declare with the .PART pseudo-op. You can choose the attributes for this partition yourself; however, all user-defined partitions must reside in NREL (locations 256 to $2^{31}-1$. See the .PART description in Chapter 7 for more information).

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 a number of partitions since all words in a module rarely have the same attributes.

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

Absolute code does not reside in a partition, but is scattered throughout the logical address space. You use the .LOC pseudo-op to place sections of this code wherever needed. Absolute code has the common base and overwrite-with-message attributes.

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 of absolute code, 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 beginning 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.

Note that each user-defined partition (i.e., defined with .PART) receives a unique relocation base.

Table 3-2 illustrates how the assembler assigns addresses to the words in your source module. By default, each data entry occupies two words of memory. Thus, there are two addresses for each data statement in Table 3-2.

Table 3-2 Assigning Addresses Within Partitions

Source Code		Addresses	Partition
.TITL	RELOC		
.ZREL			
20		0	ZREL
		1	
30		2	
		3	
.NREL	0		
40		0	Unshared NREL code
		1	
50		2	
		3	
60		4	
		5	
.NREL	1		
70		0	Shared NREL code
		1	
100		2	
		3	
.LOC	250		
110		250	Absolute code
		251	
120		252	
		253	
.NREL	0		
130		6	Unshared NREL code
		7	
.END			

During your assembly, you can leave a partition and later return to it (as in the case of the unshared NREL code partition in Table 3-2). When you return, the Macroassembler continues assigning addresses from the point where it left off earlier.

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 3-3 shows two separately assembled modules that both place code in the ZREL partition.

Table 3-3 Separately Assembled Modules With Similar Partitions

Source Module A	Relative Locations A	Source Module B	Relative Locations B
TITL A		.TITL B	
.ZREL		.ZREL	
20	0	40	0
	1		1
30	2	50	2
	3		3
.END		.END	

The Link utility can assign a value to each partition's relocation base because it knows of all the partitions and locations you use in the program. Since all predefined partitions have the normal base attribute (see Table 3-1), Link assigns values to the relocation bases in such a way that similar predefined partitions from different modules are contiguous in memory. Figure 3-5 illustrates how Link assigns addresses to similar predefined partitions from the separately assembled modules A, B, and C.

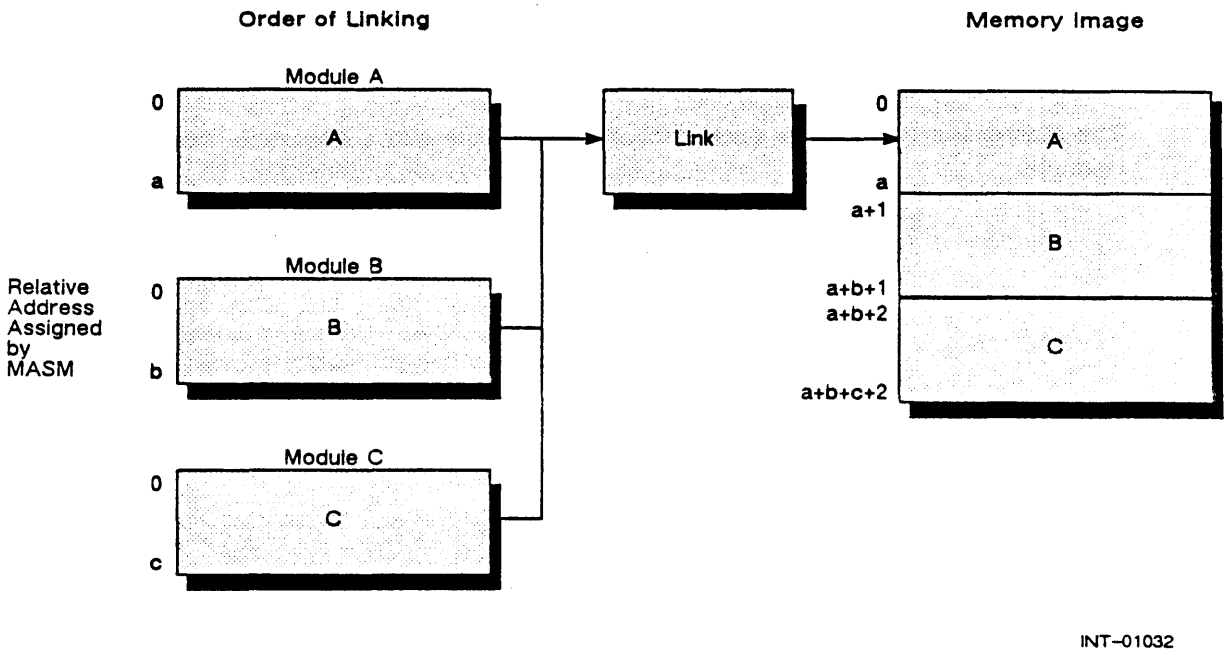


Figure 3-5 Linking Modules With Similar Predefined Partitions

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/VS Link and Library File Editor (LFE) User's Manual* provides 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 seven major relocation bases: one is associated with absolute code and the rest are associated with the six predefined memory partitions. These seven relocation bases are:

- absolute
- ZREL
- relocatable NREL data (Short NREL)
- unshared NREL code
- shared NREL code
- unshared NREL data
- shared NREL data

By using .PART, you can create additional NREL partitions. The Macroassembler assigns each user-defined partition a unique relocation base. In addition, each symbol that you declare as external (in an .EXTD, .EXTN, or .EXTL 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 predefined ZREL partition.
A:          10                      ;Each entry requires two words of
B:          20                      ;storage.  Thus, the value 10 resides
                                    ;at relative location 0 and the value
                                    ;20 resides at relative location 2.
```

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

$$B = RB_Z + 2$$

where:

RB_Z	is the ZREL relocation base
2	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 a .PART, .EXTD, .EXTN, or .EXTL 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 code partition.
X:      5               ;X has the same relocation base as
                        ;the unshared NREL code partition.
        A=.+2           ;A is defined with respect to the
                        ;location counter and, thus, has the
                        ;same relocation base as the unshared
                        ;NREL code 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 code partition.
X:      5               ;X has the unshared NREL code 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=XWLDA         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 addresses and absolute values. A symbol with an *absolute value* has a constant, integer value that remains the same through the assembly, Link, and runtime processes. The symbol's value 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 100^{TH} 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.

The address 100 is absolute only within your logical address space. That is, all processes have a location 100 and A's value is relative to your own particular address space.

Since MASM cannot determine where in AOS/VS memory your logical address space will reside, it cannot determine what A's value will be at runtime. 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 of absolute code. 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 of this manual 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 . 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:

$$A = RB_Z + 0$$

$$B = RB_Z + 2$$

where:

RB_Z is the ZREL relocation base

0 is the offset from RB_Z that MASM assigns to A

2 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:

$$\begin{aligned}(B-A)+40 &= ((RB_Z + 2) - (RB_Z + 0)) + 40 \\ &= (RB_Z - RB_Z) + (2 - 0) + 40 \\ &= (0) + (2) + 40 \\ &= 42\end{aligned}$$

In this expression, the two relocatable components (RB_Z) cancel each other out, leaving an absolute value (i.e., 42). 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 can 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
```

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 syntaxes:

<2*> **rel-symbol** { + / - } **abs-expr**

or

abs-expr { + / - } **rel-symbol** <*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 code
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 can 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 code); 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:

```
.EXTL       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 can never use two external symbols in the same expression.

The relocatable expression syntax shows that you can multiply the relocatable symbol by two. For example:

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

Both 2*X and (10)+(2*Y) are legal expressions.

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 *ECLIPSE® MV/Family 32-Bit Principles of Operations* 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 code.
X:          10                          ;Symbol X is relocatable.
            .NREL           1           ;Shared NREL code.
            XWLDA           0,X+50      ;Load the value starting at the 50th
                                        ;word after address X into AC0.
```

The XWLDA instruction provides a 16-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 16-bit field.

In short, 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).

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 2.
```

In this example, the value of label X equals the ZREL relocation base plus 2 words; i.e., X's value equals the third address in the ZREL partition. Thus, we can think of X's value as

$$X = RB_Z + 2$$

where:

RB_Z is the ZREL relocation base

2 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 expresssion as follows:

$$(X+4) = ((RB_Z + 2) + 4)$$

$$= (RB_Z + 6)$$

At this point, the Macroasembler cannot process the expression any further. Thus, it passes Link the absolute value 6 (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 code base. The Macroassembler evaluates this expression as follows:

$$(X-Y)+Z = ((RB_Z + 2) - (RB_Z + 0)) + (RB_N + 2)$$

$$= ((RB_Z - RB_Z) + (2 - 0)) + (RB_N + 2)$$

$$= ((0) + (2)) + (RB_N + 2)$$

$$= RB_N + 4$$

RB_Z is the ZREL relocation base, and RB_N is the unshared NREL code base. The values 2, 0, and 2 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 4 (integer component), the relocatable value RB_N (relocatable component), and the unresolved operator +.

The previous section explained how you can 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:

$$2*X = 2 * (RB_Z + 2)$$

$$= (2 * RB_Z) + (2 * 2)$$

$$= (2 * RB_Z) + 4$$

MASM cannot process this expression any further. Thus, it passes Link the relocatable component $2 * RB_Z$, the unresolved operator +, and the integer value 4. Any expression whose relocatable component equals two times a relocation base is byte-relocatable.

Table 3-4 shows the different forms of expressions and how the assembler resolves each.

Table 3-4 Relocatable Expressions

Expression	Relocatable Component	Integer Component	
n+m	0	n+m	
n-m	0	n-m	
n*m	0	n*m	
n/m	0	n/m	
n&m	0	n&m	
n!m	0	n!m	
d+n	base(d)	offset(d)+n	
d-n	base(d)	offset(d)-n	
d*n	base(d)*n	0	*
d/n		illegal	
d&n		illegal	
d!n		illegal	
n+d	base(d)	n+offset(d)	
n-d	-base(d)	n-offset(d)	
n*d	base(d)*n	0	*
n/d		illegal	
n&d		illegal	
n!d		illegal	
r+n	base(r)	offset(r)+n	
r-n	base(r)	offset(r)-n	
r*n	base(r)*n	offset(r)*n	**
r/n		illegal	
r&n		illegal	
r!n		illegal	
n+r	base(r)	offset(r)+n	
n-r		illegal	
n*r	base(r)*n	offset(r)*n	**
n/r		illegal	
n&r		illegal	
n!r		illegal	
r+p		illegal	
r-p	base(r)-base(p)	offset(r)-offset(p)	***
r*p		illegal	
r/p		illegal	
r&p		illegal	
r!p		illegal	

*	If offset(d) is not equal to 0, this is an error.
**	If n is not equal to either 2 or 16., this is an error.
***	If the relocation bases of r and p are the same, then the result is an absolute value If the relocation bases of r and p are not the same, and the relocation base of p is not the current partition, this is an error.
m and n	represent two distinct absolute values.
r and p	represent two distinct relocatable values.
d	is a data external, an expression involving a data external, or a symbol value defined relative to a data external. (A data external is one declared with either the .EXTDD, .EXTND or .EXTLD pseudo-op.)

Resolving Locations in Memory Reference Instructions

The section "Assembly Language Instructions" in Chapter 2 briefly mentioned memory reference instructions (MRIs) and their major uses. Again, MRIs allow you to access locations in your logical address space.

Each MRI requires you to specify a unique location in memory. You can 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 of the manual. Refer to "At Sign (@)" in Chapter 2 for a description of how MASM assembles memory reference instructions containing the indirect addressing indicator @. The *ECLIPSE® MV/Family 32-Bit Principles of Operation* manual provides more information on indirect addressing and also describes each memory reference instruction in detail.

Supplying Both a Displacement and an Index

The MV/Family 32-bit memory reference instructions provide different methods for addressing locations in memory:

- absolute addressing
- program counter (PC) relative addressing
- accumulator (AC) relative addressing

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 as an offset from the address of the MRI. That is, the Macroassembler computes the memory address by adding the displacement value to the address of the MRI.

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 can 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 3-5 describes the four index values.

Table 3-5 MRI Index Values

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

The following two examples illustrate the use of the index argument in MRIs.

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

```
XWLDA           0,3,1
100
200
```

The XWLDA 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 XWLDA instruction is two words long and the second word contains the displacement field. The value 200 begins three words after the displacement field. Therefore, after the XWLDA instruction, AC0 contains the value 200. (Remember that each data entry occupies two words of memory, by default.)

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

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

The LWLDA 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.

When specifying a displacement, be sure that value can fit into the corresponding field of the MRI. The various instructions have displacement fields of varying length (i.e., 8, 16, and 32 bits). Table 3-6 shows the legal range of the displacement value under various conditions.

Table 3-6 MRI Displacement Values

Index Value	Length of Displacement Field		
	8 bits	16 bits	32 bits *
0 (absolute addressing)	0 to 377_8 or 0 to 255_{10}	0 to 77777_8 or 0 to $32,767_{10}$	0 to 1777777777_8 or 0 to $268,435,456_{10}$
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 $+16,383_{10}$	-1000000000 to $+777777777_8$ or $-134,217,728$ to $+134,217,727_{10}$

* Note that you can refer to any location in your logical address space with a 32-bit displacement field.

Supplying Only a Displacement Value

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

The following shows how MASM generates PC-relative (index mode 1) addressing for an instruction that does not have an addressing index. The source contains:

```
            .NREL
            .
            .
A:          50
            .
            .
            LDA         0,A
            .
            .
```

MASM calculates the displacement of the LDA instruction relative to A. The displacement is simply the result of (A-.), which in this case is a negative result. The resulting instruction is the same as if the statement had been:

```
            LDA         0,A-.,1
```


In most cases, AOS/VS MASM will use PC-relative addressing if you do not supply an addressing index. There are, however, many ways that you can explicitly or implicitly override this.

- If the address is nonrelocatable, the index mode is always absolute. For example:

```
.DUSR           .FP=41
LDA             3,FP
```

  In this case, the addressing index as assembled would be 0.

- If the address is in the range 0 to 377 (octal), the index mode is always absolute. For example:

```
.LOC            41
.DUSR           FP=.
.ZREL
.DUSR           A=.
                .
                .
LDA             3,FP
LDA             2,A
```

  In both these cases, the index mode as assembled would be 0.

- If the address is in the same partition as the instruction, the index mode is always PC-relative. For example:

```
.NREL
.DUSR   A=.
LDA     0,A
```

  In this case, the index mode as assembled would be 1.

- If the instruction is LDA, STA, ISZ, DSZ, JMP, JSR, ELDB or ESTB, the index mode is absolute. For example,

```
.EXTN           A
ELDB            0,2*A
```

  In this case, the index mode as assembled would be 0.

- The default index mode as set by the .ENABLE pseudo-op, if set.
- Absolute (index mode 0) if the /W command-line switch is set,
- PC-relative (index mode 1).

Using Literals in Memory Reference Instructions

The MV/Family instruction set includes a number of instructions for loading a constant directly into an accumulator. However, the 16-bit ECLIPSE and especially the NOVA instruction set do not include similar instructions. Therefore, the assembler provides a literal facility, which manages the content and placement of groups of constants, called literal pools.

A typical memory reference instruction might look like:

```
        LDA     1,A
```

Which loads the contents of variable A into AC1. However, if you wanted to load AC1 with the constant 2, a code sequence such as the following would be required:

```
        LDA     1,C2
        .
        .
C2:     2
```

Rather than have to define label C2 and set the value of the location to 2, the assembler's literal facility can do this for you. The example above might merely be replaced with:

```
        LDA     1,=2
```

Then, when MASM assembles the instruction, "=2" is replaced with the actual address of 2 in the literal pool, which is created by the assembler.

Expressions are also acceptable as literals. For example:

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

loads AC1 with the value 100040 (octal). Furthermore, literals need not be absolute; they can be any valid expression, absolute or relocatable. You can use the literal facility to load addresses or byte addresses:

```
        LDA     1,=2*A
        .
        .
A:      .TXT    "TEXT"
```

By default, all literals are placed in a single literal pool in ZREL. Frequently, it is desirable to place literal pools elsewhere. The .NLIT pseudo-op causes the default literal pool to be placed in whatever partition is active at the end of the assembly pass. The .LPOOL pseudo-op permits creation of any number of literal pools at any place desired. Be sure that the program's flow of control does not inadvertently enter a literal pool.

End of Chapter

4

Output from the Macroassembler

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

- object file
- assembly and cross-reference listing
- error listing
- permanent symbol file

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

The *permanent symbol file* defines symbols for use in future assemblies. It usually resides in disk file MASM.PS and contains definitions for all AOS/VS system calls and system parameters.

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

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

Object File

The output file of the Macroassembler is called an *object file*. When MASM creates this object file, you are one step closer to having an executable program. When you use the AOS/VS Link utility (LINK.PR) to combine the contents of this object file with others, you produce an executable *program file* (usually designated by a .PR extension).

The contents of an object file are a representation of what was in your source file. All the code and data you defined is in it, as well as much additional information. This object file is not, however, in a human-readable form. Instead, it is in a form that is easily read by Link and other utilities. You cannot type out an object file. Symbols have been replaced with numerical values, instructions have been assembled into numerical codes, and there are no comments. In short, the object file is a machine-usable representation of your source file.

The Macroassembler normally produces an object file whether your source contains errors or not. If you do not want the Macroassembler to produce an object file, include the /N function switch on the MASM command line. Then, the Macroassembler will perform all assembly operations correctly but will not store the resulting binary object code on disk. 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 /O= 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 Hierarchy

Priority	Object Filename	Description
1 (highest)	/O=filename	Function switch on the MASM command line
2	.OB□filename	Pseudo-op in a source module
3 (lowest)	Default name	The 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/VS Link and Library File Editor (LFE) User's Manual* describes the Link utility in detail.

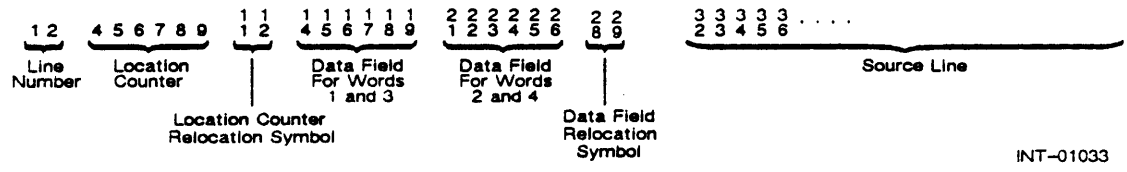
Assembly Listing

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

Table 4-2 lists the fields and the information contained in each one. Figure 4-1 is a sample assembly listing.

```
01                                  NXDGT: RCHAR 0
02
03                                      .PUSH     .NOMAC
04
05              00000000001             .NOMAC    1         ;INHIBIT LISTING
06 000056 UC 102551                   WCLM 0, 0
07 000057 UC  00000000060             "0
08 000061 UC  00000000071             "9
09 000063 UC 100570                   WBR NODIGIT
10 000064 UC 163371 177720            WNADI -"0, 0
11 000066 UC 163531                   WMULS
12 000067 UC 170770                   WBR  NXDGT
13 000070 UC 163311 000012          NODIGIT: WSEQI 12, 0
14 000072 UC 100670                   WBR  CPRINT
15 000073 UC 161431 077770          DONE:  XWSTA 0, ?0AC0, 3
16 000075 UC 165431 077772            XWSTA 1, ?0AC1, 3
17 000077 UC 103651                   WRTN
18 000100 UC 147371                 CPRINT: LPEFB ERROR*2
19              37777777576
20 000103 UC 123311                   LCALL PRINT, 0 , 1
21              00000000000  PR
22                   000001
23 000107 UC 127331                   LJMP ERRRTN
24              17777777723
25                                    .END
```

```
16 000075 UC 165431 077772          XWSTA 1, ?0AC1, 3
17 000077 UC 103651                 WRTN
18 000100 UC 147371               CPRINT: LPEFB ERROR*2
19              37777777576
20 000103 UC 123311                 LCALL PRINT, 0 , 1
21              00000000000  PR
```



INT-01033

Figure 4-1 Sample Assembly Listing

Table 4-2 Assembly Listing Fields

Columns	Information Contained
1-2	Line number within the current listing page. Each page of the listing begins with line 1.
3	Space
4-9	The value of the location counter (address), if applicable. If the current source statement generates more than one word of memory, the value in columns 4 through 9 is the address of the first one. If the current source statement does not generate any storage words, columns 4 through 9 are blank.
10	Space
11-12	A 2-character symbol indicating the relocation base (or partition) of the address in columns 4 through 9 (see Table 4-3).
13	Space
14-19	Data field for the first (or third) 16-bit word of the assembled instruction or expression; or the first (or third) 16 bits of the value on the right side of an assignment statement or a pseudo-op argument. In all other cases, these columns are blank.
20	Space
21-26	Data field for the second (or fourth) 16-bit word of the assembled instruction or expression; or the second (or fourth) 16 bits of the value on the right side of an assignment statement or a pseudo-op argument. In all other cases, these columns are blank.
27	Space
28-29	A 2-character symbol indicating the relocation base of the values in columns 14 through 19 and 21 through 26, if present (see Table 4-3).
30-31	Space
32...	The source statement exactly as written (except for macro expansions).

The Macroassembler displays the value of the location counter in columns 4 through 9. This address corresponds to the first word of the assembled source statement.

Columns 10 and 11 contain a 2-character symbol indicating the relocation base (partition) of the address. Table 4-3 lists the symbols and their meanings.

Table 4-3 Assembly Listing Relocation Symbols

Symbol	Relocation Base
(spaces)	Absolute
ZR	Lower page zero relocatable (ZREL)
LC	Short NREL
UC	Unshared relocatable code (NREL)
UD	Unshared relocatable data (NREL)
SC	Shared relocatable code (NREL)
SD	Shared relocatable data (NREL)
other	User-defined partition or external

If the address resides in a user-defined partition, the Macroassembler places the first two letters of the partition's name in columns 11 and 12. See the .PART description in Chapter 7 for more information.

Similarly, if the address value is relative to an external symbol, MASM places the first two letters of that symbol in columns 11 and 12 of the listing.

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 are two 6-column data value fields or one 11-column field. These fields contain the assembled values of the first and second 16-bit words in the current source statement. The following example shows how the MASM listing represents one- and two-word instructions:

```
01 000000    107000                  ADD   0,1      ;One-word instruction.
02 000001    163770  000100          ADDI  100,0    ;Two-word instruction.
```

If a source statement requires more than two words of memory, the assembled values of these additional words appear on subsequent lines of the listing. For example:

```
01 000000    103211  000000          WADDI   0,0      ;Three-word instruction.
02           000000
03 000003    123311  000000          LCALL   15,0     ;Four-word instruction.
04           000015  000000
```

Note that the listing contains the address of the instruction's first word only (in columns 4 through 9). In the example, the first word of the LCALL instruction resides at location 3. Subsequent words in the LCALL instruction reside at locations 4, 5, and 6.

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

```
01           000000  000001          .NREL  1
```

The first data field (columns 14 through 19) contains the value of the argument to .NREL (i.e., 1).

A 2-character symbol indicating the relocation base of the data value(s) follows the second data field. The data field relocation symbols are the same as the address relocation symbols (see Table 4-3). Chapter 3 describes how the Macroassembler assigns a relocation base to a value in your source.

If your source statement is a memory reference instruction with an externally defined displacement field, MASM places the first two letters of the external symbol in columns 28 and 29 of your listing. If the displacement field contains more than one external symbol, MASM places the first two letters of the first external symbol in columns 28 and 29.

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, and "\" and "$" numbers are expanded.

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 all assembly errors at the end of the assembly listing file. Refer to "Error Listing" later in this chapter for more information about how the Macroassembler reports assembly errors. Appendix C lists assembly errors.

Assembly Listing Control

The Macroassembler does not automatically produce an assembly listing. 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 several tools that allow you to manipulate the contents and format of the assembly listing:

listing control pseudo-ops:	.EJECT, .LCNS, .NOCON, .NOLOC, .NOMAC, and .RDXO
listing control switches:	/FF, /Z, /XREF
listing suppression indicator:	two asterisks (**)
listing suppression override switch:	/XPAND
assembly statistics switch:	/STATISTICS
page size switches:	/CPL= and /LPP=

These features do not alter the object file; they affect only assembly listing.

Listing Control Pseudo-Ops

There are five pseudo-ops that alter the assembly listing. Table 4-4 lists these pseudo-ops and describes their functions.

Table 4-4 Assembly Listing Control Pseudo-Ops

Pseudo-Op	Description
.EJECT	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)
.LNCS	Adds Data General proprietary software notice to top of each page of output

Refer to the individual pseudo-op descriptions in Chapter 7 for more detailed information about these pseudo-ops.

Asterisks (**)

You can suppress the listing of any line in the source file by placing two consecutive asterisks (**) at the beginning of that line. For example, the following source code:

```
      .NREL
      LDA     0,0,1
**    LDA     0,0,2
      LDA     0,0,3
      .END
```

produces the following assembly listing:

```
SOURCE: .MAIN              MASM 06.00.00.00       28-JAN-87 18:35:26  PAGE     1

01                                                .NREL
02 000000 UC 020400                               LDA     0,0,1
03 000002 UC 021400                               LDA     0,0,3
04                                                .END

NO ASSEMBLY ERRORS
```

Note that the location counter in the assembly listing (columns 4 through 9) jumps from 0 to 2. 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, prints the line, and reports the error.

You can override the ** listing suppression indicator at assembly time by using the /XPAND switch on the MASM command line (see Chapter 8).

Assembly Statistics

The Macroassembler reports certain statistics about your assembly in the listing file if you include the /STATISTICS switch on the MASM command line. The statistics you receive include the following:

- Total number of source lines that MASM processed
- Total elapsed time for the assembly (not CPU time)
- The average number of lines that MASM processed each minute
- The total number of symbols you used in your source module, excluding pseudo-ops and MV/Family 32-bit instruction mnemonics; that is, the statistics listing shows how many symbols MASM entered in the temporary symbol table.

MASM places the statistics information on the last page of the assembly listing. Thus, you must issue the /L or /L= function switch in conjunction with /STATISTICS. Figure 4-2 shows the statistics section from a sample assembly listing.

```
STAT:   numin              MASM 06.00.00.00       27-JAN-87 09:26:53  PAGE     4

ELAPSED TIME:   00:00:06
CPU TIME:       00:00:00
TOTAL LINES:          65
TOTAL SYMBOLS:        13
LINES/MINUTE:        650

NO ASSEMBLY ERRORS
```

Figure 4-2 Sample Statistics Listing

Page Size

By default, the Macroassembler places 80 characters on each line and 60 lines on each page of your output listing(s). These values are appropriate for 8 by 11 inch line-printer paper. Using the /CPL= and /LPP= command line switches, you can alter the size of your listing pages.

The /CPL= switch specifies how many characters the Macroassembler will place on each line of your listing. You can select any line length from 80 to 136 characters, inclusive. If a line is too long, the Macroassembler truncates the excess characters.

Similarly, the /LPP= switch indicates how many lines each listing page will contain. You can specify any value from 6 to 144, inclusive.

Please note that the /CPL= and /LPP= function switches apply to the error and cross-reference listings, as well as to the assembly listing.

Refer to Chapter 8 for more information about command line function switches.

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 4-5 lists the assignment mnemonics and their meanings.

Table 4-5 Cross-Reference Assignment Mnemonics

Mnemonic	Meaning	Defining Pseudo-op
EN	Entry symbol	.ENT
MC	Macro symbol	.MACRO
NC	Named common symbol	.COMM
XD	External displacement symbol	.EXTD
XN	External narrow symbol	.EXTN
XL	External long symbol	.EXTL
PT	User-defined partition symbol	.PART
(spaces)	All other symbols	

These mnemonics appear in the cross reference immediately after the symbol's value (where applicable).

Figure 4-3 is a sample cross-reference listing.

```
XREF:   numin              MASM 06.00.00.00       27-JAN-87 09:26:53  PAGE    3
?OAC0      37777777770       2/15
?OAC1      37777777772       2/16
?READ      00000000000 MA    2/06
?SYST      00000000000 MA    2/06
?XCALL     00000000001       2/06     2/06
BUF$       00000000000 XL    2/06
CPRINT     00000000100       2/14     2/18#
DONE       00000000073       2/15#
ERROR      00000000000       1/07#    2/18
ERRRTN     00000000033       1/59#    2/23
IO$DEF     00000000000       1/21#
IO$ERR     00000000000 XL    2/06
IO$PKT     00000000000 XL    2/06
NODIGIT    00000000070       2/09     2/13#
NUMIN      00000000031 EN    1/04     1/58#
NXDGT      00000000036       2/01#    2/12
PRINT      00000000000 XL    1/03     2/20
RCHAR      00000000000 MA    1/23#    2/01
```

Figure 4-3 Sample Cross-Reference Listing

Cross-Reference Listing Control

The assembler produces a cross-reference listing only when it generates an assembly listing. Thus, to receive a cross-reference listing, you must include the /L or /L= switch on the MASM command line. The cross reference appears after the assembly listing in the same file.

The cross-reference listing normally contains only numeric symbols. By using the /XREF= switch on the MASM command line, you can direct the Macroassembler to include macro, instruction, and permanent symbols in the listing. The /XREF= switch also allows you to suppress the cross-reference listing completely.

Chapter 8 describes the various values you can pass to the /XREF= switch. Chapter 2 explains the four symbol types (numeric, macro, instruction, and permanent).

Error Listing

The *error listing* contains the title of your source module and information about all statements in it that cause errors. MASM divides the error listing into two sections:

- Pass one errors
- Pass two errors

The first section lists errors that MASM detected on its first pass through your program. The *pass one error section* includes the following information:

- The source page and line numbers of all statements that cause pass one errors
- A short message describing each assembly error (Appendix C lists all assembly error messages)

The second section of the listing shows the errors that MASM detected on its second pass through your program. This *pass two error section* includes the following:

- The page and line numbers in your source module for all statements that cause pass two errors
- The page and line numbers in your assembly listing for all statements that cause pass two errors; MASM includes this information only if you produce an assembly listing (i.e., issue the /L or /L= function switch on the MASM command line)
- A short message describing each assembly error (see Appendix C)

The Macroassembler may report many errors for the same source statement; there is no maximum. In addition, a source statement may generate the same error on both assembler passes.

Figure 4-4 is a sample error listing. The Macroassembler generated this listing while assembling source file MODULE1.SR, which contains the following source code:

```
                .TITLE          MOD1
                .NREL
                X=9
                B=DATA+2
    DATA:       50
                .END


ERRORS: MOD1                MASM 06.00.00.00        28-JAN-87 18:56:41  PAGE    3
pass 1 errors:
 SOURCE     1/3: error in a constant
 SOURCE     1/4: symbol is undefined
pass 2 errors:
 SOURCE     1/3: listing   1/03: error in a constant
3 ASSEMBLY ERRORS
```

Figure 4-4 Sample Error Listing

Figure 4-4 shows that the assembler detected a string conversion error on pass one. The report indicates that the third statement in the first page of the source module causes that error.

```
X=9
```

Since the default input radix is octal (base 8), the digit 9 is illegal.

The assembler also reported an undefined symbol error on the first pass for the statement

```
B=DATA+2
```

The source code in MODULE1.SR does not define the symbol DATA until after this statement. Thus, on pass one, MASM did not have a definition for symbol DATA.

On pass two, MASM again detected the string conversion error (illegal digit for octal input). However, the undefined symbol error disappeared. That is, on pass two, MASM had information about the symbol DATA and, therefore, did not return an error for the statement

```
B=DATA+2
```

Since MASM may resolve pass one errors by the end of the assembly, you usually use the pass two error section to locate errors in your source.

Appendix C lists the assembly errors that the Macroassembler can 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. Within the assembly listing, the error report appears after the source code listing.

Output Function Switches

Throughout this chapter, we have discussed the MASM command line switches that affect Macroassembler output. Table 4-6 lists useful switch combinations, shows what output the Macroassembler produces in each case, and also indicates where that output resides.

Table 4-6 Macroassembler Output Function Switches

Output Function Switches	Output				
	Assembly Listing	Cross-Reference Listing	Error Reports	Object File	Permanent Symbol File
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		
/O=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=FILE2 /E=FILE2 /O=FILE3	FILE1	FILE1	FILE1 and FILE2	FILE3.OB	
/MAKEPS			@OUTPUT		MASM.PS

The general form for using these switches on the MASM command line is

XEQ MASM<*function switch*>... sourcefile</*PASS1*> ...

Refer to Chapter 8 for descriptions of /PASS1 and all the MASM function switches and for further discussion of the Macroassembler command line.

End of Chapter

5

Macros and Generated Numbers and Symbols

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 can assign the source string a name. Then, each time you want 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 must be the first character on that source line. Do not place any spaces or tabs before the percent character.

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 137_8) 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 at the beginning of 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 can 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 the underscore. For example, the assembler interprets

```
          .MACRO        X
          A_B                             ;The assembler removes_
%                                         ;from the symbol A_B.
```

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 can 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.

To delete a macro definition, use the .XPNG pseudo-op. .XPNG removes the macro name and the associated macro definition string. If you try to redefine a macro name without first using .XPNG, the Macroassembler returns an error. See .XPNG in Chapter 7 for more information.

Arguments in Macro Definitions

You can 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 conventional 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 macro, it replaces all occurrences of ^n with the n^{th} 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 and, 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}.

In order to make use of the ^a or ^?a formats, you must assign numeric values to the symbols used in the formats. By convention, you assign the value 10 to the letter A in the ^a format, so that MASM replaces ^A by the tenth argument in the macro call. Similarly, the convention calls for you to assign B the value 11, C the value 12, and so on. The convention is illustrated in the table below.

Symbol	Value
A	10
B	11
C	12
D	13
E	14
.	.
.	.
.	.
V	31
W	32
X	33
Y	34
Z	35

The a or ?a following ^ is a numeric 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. Note that this example does not follow the convention above, which is needed only for macros with more than nine arguments.

```
        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 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 can 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 available for passing arguments to macros:

1. **macro-name**□**arg...**

2. **macro-name**□**[arg...** ↵
   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**
↵	is a New Line or Carriage Return you must enter

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 can 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 illegal for an 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, 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 can *never* pass certain characters as arguments in a macro call (even within square brackets). These characters include space, horizontal tab, comma, carriage return, form feed, NEW LINE, left bracket ([), and right bracket (]).

Null Arguments

In certain situations, you may want 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
          LWLDA         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 can indicate indirect addressing by supplying the character @, or you can pass a null string.

Our first call to macro ADDR passes @.

```
ADDR            [@],LOC1,2
```

This macro call generates the source statement LWLDA 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 LWLDA 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. Again, note that the assembly listing contains both the macro call DSP 2 and the expansion 2.

Source Text

```
        .MACRO  DSP     .;Macro definition.
          ^1
%
          DSP     2       ;Call to macro DSP.
```

Assembly Listing

```
01                                      .MACRO  DSP     ;Macro definition.
02                                      ^1
03                                  %
04
05                                      DSP     2       ;Call to macro DSP.
06 000000       00000000002             2
```

Figure 5-1 Macro Listing

The assembly listing values for the location counter (columns 4-9) and the data fields (columns 14-19 and 21-26) reveal 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 can 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.

```
01                                      .MACRO  Z       ;Define macro Z.
02                                      5
03                                      LDA     ^1,^2
04                                %
05
06                                                      ;Macro expansions are
07                                      Z       0,4     ;listed by default.
08 000000     00000000005               5
09 000002    020004                     LDA     0,4
10
11             00000000001              .NOMAC  1       ;Directs the assembler to
12                                                      ;suppress listings of
13                                                      ;macro expansions (i.e., pass
14                                                      ;a nonzero value to .NOMAC).
15                                      Z       0,4
16
17             00000000000              .NOMAC  0       ;Re-enable the listing of macro
18                                                      ;expansions (i.e., pass a zero
19                                                      ;value to .NOMAC).
20                                      Z       0,4
21 000006     00000000005               5
22 000010    020004                     LDA     0,4
23                                      .END
```

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 2 to 6 in the above listing example; the missing locations represent macro expansions whose listings were suppressed.

You can override the .NOMAC pseudo-op at assembly time by using the /XPAND switch on 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 can 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 000000     00000000011               4+5
08 000002     00000000002               (.ARGCT)
09                                                      ;At expansion time, the value for
10                                                      ;.ARGCT is 2 because macro X was
11                                                      ;called with two arguments.
```

The .MCALL pseudo-op is also a value symbol that you can 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 .NOMAC.

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. The assembler reports an error if it encounters the macro definition terminator % before .ENDC. In addition, MASM takes one of the following actions:

- 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
%
```

When you call macro ERRDO, the Macroassembler will report an error and ignore the .DO statement since it is not terminated inside the macro (i.e., no corresponding .ENDC).

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.
%
```

This is an example of an *incorrect* .IF conditional inside a macro:

```
        .MACRO      ERRIF
        .IFE        ^1              ;Incorrect use of .IFE in a macro.
            50
            60
%
```

When you call macro ERRIF, the Macroassembler reports an error since the .IFE conditional is not terminated inside the macro (no .ENDC). In this case, 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 WIOR instruction (see Chapter 8) to perform the logical OR operation; we present this example only to illustrate the macro facility.

```
        .MACRO      OR
        WCOM        ^1,^1           ;Complement AC^1.
        WAND        ^1,^2           ;Clear ON bits of AC^1.
        WADC        ^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.

```
07                                      OR      1,2
08 000000    126131                     WCOM    1,1    ;Complement AC1.
09 000001    132111                     WAND    1,2    ;Clear ON bits of AC1.
10 000002    131111                     WADC    1,2    ;OR result to AC2.
```

Example 2: IF-THEN-ELSE

Our next example uses a macro to implement an IF-THEN-ELSE structure. Macro IF compares two accumulators and, depending on the outcome, transfers control to one of two specified addresses.

The format for calling macro IF is

IF□ac_1□ac_2□test□$addr_t$□$addr_f$

where:

IF	is the name of the macro
ac_1	is the first accumulator
ac_2	is the second accumulator
test	is a number, from 0 to 3, that indicates one of the following tests:

Value	Test Performed
0	Is $ac_1 > ac_2$?
1	Is $ac_1 = ac_2$?
2	Is $ac_1 < ac_2$?
3	Is $ac_1 <> ac_2$?

$addr_t$	is the address the macro returns to if the test condition is true
$addr_f$	is the address the macro returns to if the test condition is false

As an example, consider the following macro call:

```
IF      1,2,0,LOC1,LOC2
```

This macro expansion tests to see if accumulator 1 is greater than accumulator 2. If so, program execution continues at LOC1; otherwise, control passes to address LOC2.

The macro definition for IF follows:

```
                .MACRO          IF

                .IFE            .MCALL          ;Variable COUNT keeps track of how
                COUNT=0                         ;many times you call the macro IF.  This
                .ELSE                           ;variable is necessary to generate
                COUNT=COUNT+1                   ;unique labels each time you call macro
                .ENDC                           ;IF (see the last two lines in this
                                                ;macro).

                .IFN            ^3==0           ;GREATER THAN test (third argument to
                WSGT            ^1,^2           ;IF is 0).  If the first AC is greater
                WBR             FALSE\COUNT     ;than the second, go to location ^4;
                WBR             TRUE\COUNT      ;else go to ^5.
                .ENDC

                .IFN            ^3==1           ;EQUAL test (third argument to IF is 1).
                WSEQ            ^1,^2           ;If the first AC equals the second AC,
                WBR             FALSE\COUNT     ;go to location ^4; else go to ^5.
                WBR             TRUE\COUNT
                .ENDC

                .IFN            ^3==2           ;LESS THAN test (third argument to IF
                WSLT            ^1,^2           ;is 2).  If the first AC is less than
                WBR             FALSE\COUNT     ;the second, go to location ^4; else
                WBR             TRUE\COUNT      ;go to ^5.
                .ENDC

                .IFN            ^3==3           ;NOT EQUAL test (third argument to IF
                WSNE            ^1,^2           ;is 3).  If the first AC does not equal
                WBR             FALSE\COUNT     ;the second, go to location ^4; else
                WBR             TRUE\COUNT      ;go to ^5.
                .ENDC

FALSE\COUNT:                    LJMP^5          ;Jump to appropriate address (^4 or ^5).
TRUE\COUNT:                     LJMP^4          ;The backslash (\) directs MASM to create
                                                ;two new labels each time you call IF
                                                ;(see "Generated Numbers and Symbols"
                                                ;later in this chapter).

%
```

Example 3: Factorial

Our third example illustrates the recursive property of macros. Macro FACT computes factorials. 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

```
            .MACRO                          FACT
**          .DO             ^1<0
                            ^2=0
**          .ENDC
**          .DO             ^1<=1
**                          ^2=1
**          .ENDC
**          .DO             ^1<>1
                            FACT            ^1-1,^2
                            ^2=^1*^2
**          .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 less than 0, FACT returns the value 0 in variable V.
- If I is 0 or 1, FACT returns the value 1 in variable V.
- If I is greater than 1, macro FACT calls itself recursively with successively smaller values for I. When the integer argument to FACT equals 1, the second .DO conditional expands to completion. This begins a succession of returns to each level that made a recursive call to FACT. As these levels expand to completion, MASM computes I! and stores the result in V.

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 in the macro definition suppress the listing of all other macro statements.

```
                                    FACT    4,A      ;Initial call.
                                            FACT    4-1,A
                                            FACT    4-1-1,A
                                            FACT    4-1-1-1,A
                                            FACT    4-1-1-1-1,A
                                            A=4-1-1-1*A
000000 000002                               A=4-1-1*A
000000 000006                               A=4-1*A
000000 000030                               A=4*A
```

Example 4: Packed Decimal

Our last macro example stores numeric values in *packed decimal* format. In packed decimal format, 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 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 the first decimal digit.
d...	is a series of optional decimal digits. If you supply more than one digit, separate each digit with spaces, tabs, or commas.
s	is the sign of the decimal number (+ or -).
w	is the precision and indicates how any 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
                        .WORD   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:

```
35              00000000100                 .LOC    100      ;Start at address 100 (octal)
36                                                           ;or 40 (hexadecimal)
37                                          PACK    1 2 +,2
38 000041       0123                        .WORD   W
39 000040       0000                        .WORD   W
40
41                                          PACK    1 2 3 4 5 +,3
42 000044       3453                        .WORD   W
43 000043       0012                        .WORD   W
44 000042       0000                        .WORD   W
45
46                                          PACK    1 2 3 4 5 -,3
47 000047       3454                        .WORD   W
48 000046       0012                        .WORD   W
49 000045       0000                        .WORD   W
50
51                                          PACK    9 8 7 6 5 +,6
52 00004D       7653                        .WORD   W
53 00004C       0098                        .WORD   W
54 00004B       0000                        .WORD   W
55 00004A       0000                        .WORD   W
56 000049       0000                        .WORD   W
57 000048       0000                        .WORD   W
58
```

Generated Numbers and Symbols

You can 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.

The \symbol switch can stand alone in code to form an integer, or it can 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
```

The assembly listing for a generated number or symbol shows only the replacement value, not the **\symbol** designation. For example, the above section of source code would appear as follows in the assembly listing:

```
01              00000000002                 A=2             ;Initialize A and B.
02              00000001234                 B=1234
03 000000       00000000001     X002:        1               ;X\A evaluates to the symbol X002.
04 000002       00000000001     X234:        1               ;X\B evaluates to the symbol X234
                                                            ;(the "1" is truncated from "1234").
05
06              00000000236                 C=002+234         ;\A equals 002 and \B equals 234 so
                                                            ;C equals 236.
07
08 000004       10307020010                 450.002           ;450.\A evaluates to 450.002
```

Table 5-1 shows the correspondence between source code, the assembly listing, and the cross-reference listing. The assembly and cross-reference listings reflect the actual symbol names.

Table 5-1 Generated Symbols in Source and Listings

Source Code	Assembly Listing	Cross-Reference Listing
ONES=111	ONES=111	ONES
A\ONES	A111	A111

You can 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 loop 64 (decimal) times.
A\X:                    0                   ;Create labels A000, A001,...,A077.
                                            ;NOTE:label numbers in octal (77₈ =63₁₀)
**                      X=X+1               ;Increment counter X.
**          .ENDC                           ;End of .DO loop.
```

The listing for this section of code follows:

```
01              00000000010                 .RDX    8
02
03              00000000100         TABLE:  .DO     64.     ;Assemble loop 64 (decimal) times.
04 000000       00000000000         A000:           0       ;Create labels A000, A001,...,A077.
05 000002       00000000000         A001:           0       ;Create labels A000, A001,...,A077.
06 000004       00000000000         A002:           0       ;Create labels A000, A001,...,A077.
07 000006       00000000000         A003:           0       ;Create labels A000, A001,...,A077.
08 000010       00000000000         A004:           0       ;Create labels A000, A001,...,A077.
                                        .
                                        .
                                        .
SOURCE: .MAIN               MASM 06.00.00.00        28-JAN-87 08:47:12  PAGE    2

01 000162       00000000000         A071:           0       ;Create labels A000, A001,...,A077.
02 000164       00000000000         A072:           0       ;Create labels A000, A001,...,A077.
03 000166       00000000000         A073:           0       ;Create labels A000, A001,...,A077.
04 000170       00000000000         A074:           0       ;Create labels A000, A001,...,A077.
05 000172       00000000000         A075:           0       ;Create labels A000, A001,...,A077.
06 000174       00000000000         A076:           0       ;Create labels A000, A001,...,A077.
07 000176       00000000000         A077:           0       ;Create labels A000, A001,...,A077.
```

If the /$ command-line switch is set, you can use the dollar sign character ($) to generate unique symbol names within macros. Each occurrence of the character $ is replaced by three characters from the set 0-9 and A-Z. The three characters are determined by converting a count of the number of macro calls in radix 36 to ASCII characters, where A=10 (decimal) and Z=35 (decimal).

For example, each call to the following macro creates and initializes a new one-word storage location:

```
              .MACRO BLAH
              .
              .
              .
  LOC\$:      .WORD      0          ;The first call to this macro creates
              .                     ;LOC000, the second creates LOC001, the third
              .                     ;creates LOC002, and so on, providing the /$
              .                     ;command-line switch is set.
%
```

In nested macro calls, the replacement value for $ in the outer macro is saved and restored when the inner macro call has been fully expanded.

When you use $ in symbol names, $ should not be the first character in the name, since the first character it would be replaced by might be a digit.

End of Chapter

6

Types of Pseudo-Ops

Pseudo-ops direct the action of the macroassembler, rather than the user program. Pseudo-ops can generally be divided into one of the following categories:

- location pseudo-ops
- file termination pseudo-ops
- conditional assembly pseudo-ops
- macro assembly pseudo-ops
- data formatting pseudo-ops
- literal pseudo-ops
- inter-module communication pseudo-ops
- listing control pseudo-ops
- stack control pseudo-ops
- radix control pseudo-ops
- text string pseudo-ops
- symbol table pseudo-ops
- miscellaneous pseudo-ops

The following sections of this chapter describe each of these categories and the pseudo-ops in those categories.

Location Pseudo-Ops

Generally, you use location pseudo-ops (see Table 6-1) to set the place that code and/or data will occupy in the program. You can specify these locations explicitly (.LOC 14), in which case the code and/or data is absolute, otherwise the code or data is relocatable. Relocatable sections of code and/or data may be named or unnamed. You can name them by using the .PART pseudo-op, or they may be unnamed, and therefore part of a default partition, using the .NREL and .ZREL pseudo-ops.

Two pseudo-ops, .BLK and .ALIGN, increment the location counter. .BLK takes an explicit increment, while .ALIGN increments the location counter to a given memory boundary.

Both .LOC and . may be used in expressions to obtain the value and relocation property of the location counter. The results obtained from the two pseudo-ops are the same in all cases.

Table 6-1 Location Pseudo-Ops

Pseudo-op	Directive	Value	Description
.		X	Return value of location counter
.ALIGN	X		Align location to memory boundary
.BLK	X		Reserve a block of data words
.GLOC	X		Set location counter
.LOC	X	X	Set or return location counter
.NREL	X		Use default NREL partition
.PART	X		Use user-defined partition
.ZREL	X		Use default ZREL partition

File Termination Pseudo-Ops

The assembler provides three explicit file termination pseudo-ops. These pseudo-ops, listed in Table 6-2, all cause MASM to cease reading, ignoring the remainder (if any) of the input file.

.END, in addition to terminating the source file, may take an argument giving the starting address for program execution.

.EOF and .EOT are equivalent. .EOT is provided for compatibility with other Data General assemblers.

Table 6-2 File Termination Pseudo-Ops

Pseudo-op	Directive	Value	Description
.END	X		End-of-file with start address
.EOF, .EOT	X		End-of-file

Conditional Assembly Pseudo-Ops

The pseudo-ops in this category allow you to assemble a series of source lines repetitively, or to suppress their assembly entirely. The conditional assembly pseudo-ops are listed in Table 6-3.

The .DO pseudo-op causes all source lines between it and its corresponding .ENDC pseudo-op to be assembled repetitively. You specify the number of times as an argument to .DO.

The .IFE, .IFG, .IFL and .IFN pseudo-ops permit conditional assembly based on whether their argument is zero or nonzero. The .ELSE pseudo-op, when used with the .IFx pseudo-ops, permits conditional assembly of two groups of source lines in an either/or manner.

For compatibility with other Data General assemblers, both .ENDC and .GOTO can take a label argument, in which case, the following source lines are bypassed until the label, enclosed in brackets, is seen.

Table 6-3 Conditional Assembly Pseudo-Ops

Pseudo-op	Directive	Value	Description
.DO	X		Repetitively assemble conditional code
.ELSE	X		Reverse sense of conditional assembly
.ENDC	X		Terminate conditional assembly
.GOTO	X		Jump ahead in conditional assembly
.IFE	X		Assemble if argument equals zero
.IFG	X		If argument is greater than zero
.IFL	X		If argument is less than zero
.IFN	X		If argument is not equal to zero

Macro Assembly Pseudo-Ops

Pseudo-ops associated with macros are listed in Table 6-4. You define macros using the .MACRO pseudo-op. There are two value pseudo-ops for use inside macros. The .ARGCT pseudo-op returns the number of arguments that a macro was called with, while .MCALL returns a flag indicating whether the macro has been called previously.

Table 6-4 Macro Assembly Pseudo-Ops

Pseudo-op	Directive	Value	Description
.ARGCT		X	Return number of macro arguments
.MACRO	X		Define a macro
.MCALL		X	Return 1 if macro called previously

Data Formatting Pseudo-Ops

You should use data formatting pseudo-ops (listed in Table 6-5) to specify size, range of value, or relocation properties of data words. While use of the data formatting pseudo-ops is recommended, it is not required.

The .DWORD and .WORD pseudo-ops provide explicit specification of the size of a data item. .SWORD and .UWORD generate one-word data items like .WORD, but with value range checks at assembly, and, if necessary, at linkage time.

The .GATE pseudo-op permits the generation of gate array entries. Gate arrays control calls from one ring to another in the MV/Family architecture.

The remainder of the data formatting pseudo-ops are primarily for use with 16-bit programs. The .GADD and .GREF pseudo-ops are provided primarily for compatibility with other Data General assemblers. .GADD generates a relocatable data word with WORD (16-bit) relocation. .GREF generates a relocatable data word with GREF (15-bit) relocation.

The .CALL, .KCALL, .RCALL and .RCHAIN pseudo-ops generate a one-word data item with CALL relocation, for use with the AOS or AOS/VS 16-bit resource manager. .CALL requires you specify the value of the data word; the others implicitly set the value of the data word. Likewise, .TARG and .PTARG generate TARGET relocation for using the resource manager; .TARG requires the data word value, .PTARG does not.

Table 6-5 Data Formatting Pseudo-Ops

Pseudo-op	Directive	Value	Description
.CALL	X		Create data word with CALL relocation
.DWORD	X		Create two-word data items
.GADD	X		Create data word with WORD relocation
.GATE	X		Create gate entry with ring field
.GREF	X		Create data word with GREF relocation
.KCALL	X		Create data word with CALL relocation
.PTARG	X		Create data word with TARGET relocation
.RCALL	X		Create data word with CALL relocation
.RCHAIN	X		Create data word with CALL relocation
.SWORD	X		Create one-word signed data items
.TARG	X		Create data word with TARGET relocation
.UWORD	X		Create one-word unsigned data items
.WORD	X		Create one-word data items

Literal Pseudo-Ops

Literals are needed only for 16-bit programs. Literals are data words created automatically by the assembler, e.g., "LDA 1,=2" creates a literal of 2. The assembler brings literals together in groups called literal pools. There are two pseudo-ops for controlling the placement of literal pools, and these are listed in Table 6-6. If neither of the pseudo-ops is used, the assembler generates a single default literal pool, and places it in ZREL. The .NLIT pseudo-op causes this default literal pool to be placed in whatever partition is active at the end of the assembly pass. The .LPOOL pseudo-op explicitly locates literal pools, bypassing the default pool.

Table 6-6 Literal Pseudo-Ops

Pseudo-op	Directive	Value	Description
.LPOOL	X		Create a literal pool
.NLIT	X		Place default pool in NREL

Inter-module Communication Pseudo-Ops

Inter-module communication pseudo-ops allow you to define symbols and data in one source module and to refer to that information in a separately assembled module. These pseudo-ops, listed in Table 6-7, can define and refer to absolute values, relocatable values and common areas.

You can define a symbol's value either with an equate statement or as a label. The symbol's value, through its name, can be made visible to other modules with the .ASYM, .ENT or .PENT pseudo-ops. The other modules must then refer to the symbol with one of the .EXTx pseudo-ops.

The .ENT pseudo-op is the normal pseudo-op for making a symbol definition globally visible. Use .PENT only for entries into 16-bit programs that will be using the AOS or AOS/VS resource manager. .ASYM allows more than one module to define a symbol, and the symbol's value to be the sum of their definitions.

The .EXTx pseudo-ops require you to specify the size and type of the external data. To do this, replace "x" with either one, two, or three letters which describe the external. The forms of the .EXTx pseudo-op are:

$$\text{.EXT} \left\{\begin{matrix}\text{D}\\\text{N}\\\text{L}\end{matrix}\right\} \left\{\begin{matrix}\text{-}\\\text{A}\\\text{D}\end{matrix}\right\} \left\{\begin{matrix}\text{-}\\\text{N}\\\text{W}\end{matrix}\right\}$$

The first letter choice, D, N or L, specifies the expected size of the external's value: D specifies an 8-bit external, N a 16-bit external, L a 32-bit external. The second letter choice, if present, specifies whether the external is an address or an absolute value. If the second letter is not given, the external is assumed to be an address. If the second letter choice is "A", then the third letter choice can optionally specify whether the address points to a 16-bit or 32-bit data item. If not given, it can be either.

The .EXTC pseudo-op allows one or more modules to build up a chain of backward-linked memory locations. That is, each reference to a symbol defined with a .EXTC gets the value of the previous reference to that symbol.

The .ENTO pseudo-op is for use of 16-bit programs that do not use the AOS and AOS/VS 16-bit resource manager. It defines a symbol whose value is the overlay area and overlay number that the symbol is defined in.

The .COMM and .EXTU pseudo-ops are provided primarily for compatibility with other Data General assemblers. .COMM defines a common area, as does the .PART pseudo-op. .EXTU causes all undefined symbol error messages to be suppressed, and assumes undefined symbols to be externals.

Table 6-7 Inter-module Communication Pseudo-Ops

Pseudo-op	Directive	Value	Description
.ASYM	X		Define an accumulating symbol
.COMM	X		Define a named common symbol
.ENT	X		Define a global entry symbol
.ENTO	X		Define an overlay entry symbol
.EXTC	X		Define a chain-link external
.EXTD	X		Define 8-bit address external
.EXTDA	X		Define 8-bit address external
.EXTDAN	X		Define 8-bit address of one word
.EXTDAW	X		Define 8-bit address of two words
.EXTDD	X		Define 8-bit data external
.EXTG	X		Define 32-bit data external
.EXTL	X		Define 32-bit address external
.EXTLA	X		Define 32-bit address external
.EXTLAN	X		Define 32-bit address of one word
.EXTLAW	X		Define 32-bit address of two words
.EXTLD	X		Define 32-bit data external
.EXTN	X		Define 16-bit address external
.EXTNA	X		Define 16-bit address external
.EXTNAN	X		Define 16-bit address of one word
.EXTNAW	X		Define 16-bit address of two words
.EXTND	X		Define 16-bit data external
.EXTU	X		Make all undefined symbols external
.PENT	X		Define a procedure-entry symbol

Listing Control Pseudo-Ops

There are several pseudo-ops which do not affect code or data, but control only the assembly listing. These pseudo-ops are described in Table 6-8.

The .EJECT pseudo-op causes the next line after the .EJECT to unconditionally begin a new page.

The .NOCON, .NOLOC and .NOMAC pseudo-ops suppress parts of the listing. The .NOCON pseudo-op suppresses source lines which are not assembled due to a false conditional pseudo-op. The .NOLOC pseudo-op suppresses all source lines which do not have an explicit location, e.g., the second line of .TXT strings. The .NOMAC pseudo-op suppresses the listing of macro expansions.

The .ERROR pseudo-op prints a user-defined error message in the listing. The .LCNS pseudo-op places a Data General proprietary header on each page of the listing, the same as the /Z command-line switch.

Table 6-8 Listing Control Pseudo-Ops

Pseudo-op	Directive	Value	Description
.EJECT	X		Begin a new page of the listing
.ERROR	X		Report a user-defined error message
.LCNS	X		Begin listing page with DG header
.NOCON	X	X	Suppress listing of conditional code
.NOLOC	X	X	Suppress listing of noncode
.NOMAC	X	X	Suppress listing of macro expansions

Stack Control Pseudo-Ops

The Macroassembler maintains a push-down stack which can be used at assembly time to save the value and relocation property of any expression. In a push-down stack, the last expression you place on the stack is always the first to be removed. The pseudo-ops associated with this stack are listed in Table 6-9.

To place an expression on the stack, use the .PUSH pseudo-op. To get the value of the item on the top of the stack, use the .TOP pseudo-op. To get the value of the item on the top of the stack, and pop it off, use the .POP pseudo-op.

Table 6-9 Stack Control Pseudo-Ops

Pseudo-op	Directive	Value	Description
.POP		X	Return and pop item from top of stack
.PUSH	X		Push item onto top of stack
.TOP		X	Return item from top of stack

Radix Control Pseudo-Ops

The assembler allows you to specify both the input radix that the source file is interpreted in and the output radix used in listing the source. These two radices are separate and you can set them to the same or different values using the pseudo-ops listed in Table 6-10.

The default for both input and output is octal; that is, radix 8. The input radix can be set to any integer between 2 and 20 (decimal), while the output radix can be between 8 and 20 (decimal).

Table 6-10 Radix Control Pseudo-Ops

Pseudo-op	Directive	Value	Description
.RDX	X	X	Radix for numeric input
.RDXO	X	X	Radix for numeric output on listing

Text String Pseudo-Ops

There are four pseudo-ops for assembling text strings into their ASCII codes, and two additional pseudo-ops that affect how text strings are assembled. These pseudo-ops are listed in Table 6-11.

The .TXT, .TXTE, .TXTF and .TXTO pseudo-ops all take a text string as an argument. They differ only in how the parity bit (bit 0) of each byte of text is set. .TXT generates no parity. The .TXTE pseudo-op generates even-parity data. The .TXTF pseudo-op generates one-parity data. The .TXTO pseudo-op generates odd-parity data.

The .TXTM pseudo-op changes how two characters are stored in one word of memory. By default, characters are packed into words left-to-right.

The .TXTN pseudo-op enables or disables the automatic addition of a terminating null (zero) byte on all text strings.

Table 6-11 Text String Pseudo-Ops

Pseudo-op	Directive	Value	Description
.TXT	X		Create data from text string
.TXTE	X		Create even-parity data from string
.TXTF	X		Create one-parity data from string
.TXTM	X	X	Set high/low packing in text strings
.TXTN	X	X	Add null byte to text strings
.TXTO	X		Create odd-parity data from string

Symbol Table Pseudo-Ops

The majority of the symbol-table pseudo-ops (see Table 6-12) define MV/Family instructions. Since the MV/Family instructions are already included in AOS/VS MASM's permanent table, few programmers will need to use symbol-table pseudo-ops. However, programmers frequently use the .DUSR pseudo-op to define new instructions or redefine old instructions previously deleted by the .XPNG pseudo-op.

Symbol table pseudo-op statements have the form:

.Dxxx□numeric-symbol = expression

or

.Dxxx□numeric-symbol = instruction□[instruction-argument ...]

The value of the expression, or the assembled value of the instruction and its arguments, is assigned to numeric-symbol. Unlike other MASM assignment statements (A=5), the value assigned to numeric-symbol is not expected to change.

To clarify the difference between using .DUSR and using a simple assignment statement, consider the following code:

```
.TITLE FOOBAR
.NREL      1
           A = 5
.DUSR      Z = 5
```

At first glance, there doesn't seem to be any difference between A and Z. Both represent the numerical value 5. However, A is, implicitly, an assembly-time variable, while Z is an assembly-time constant. That is, the value of A may change later on in the same module, while the value of Z should remain constant throughout. If a subsequent assignment changes Z's value, MASM reports an error if the /MULTIPLE command-line switch is set.

There is an instruction format associated with each of the instruction-defining pseudo-ops. Each instruction format assumes a certain instruction size, a minimum and maximum number of arguments, and a range of values for each argument. The pseudo-op also defines whether the instruction might skip the next one-word instruction. These formats are shown in detail under the specific description of each pseudo-op.

You can use the .XPNG pseudo-op to remove any one or all instruction definitions from AOS/VS MASM's permanent symbol table. It can also be used to delete symbols from the symbol table. However, you cannot use .XPNG to delete pseudo-op symbols from the permanent symbol table.

Table 6-12 Symbol Table Pseudo-Ops

Pseudo-op	Directive	Value	Description
.DALC	X		Define an ALC (ADD) instruction
.DCMR	X		Define ELDB-format instructions
.DEMR	X		Define EJMP-format instructions
.DERA	X		Define ELEF-format instructions
.DEUR	X		Define SAVE-format instructions
.DFLM	X		Define FAMS-format instructions
.DFLS	X		Define FSST-format instructions
.DIAC	X		Define HLV-format instructions
.DICD	X		Define ADI-format instructions
.DIMD	X		Define CIOI-format instructions
.DIMM	X		Define ADDI-format instructions
.DIMS	X		Define WSEQI-format instructions
.DIO	X		Define NIO-format instructions
.DIOA	X		Define DIA-format instructions
.DISD	X		Define PSH-format instructions
.DISS	X		Define SGT-format instructions
.DIWM	X		Define WADDI-format instructions
.DIWS	X		Define WUGTI-format instructions
.DLBA	X		Define LLEFB-format instructions
.DLBR	X		Define LPEFB-format instructions
.DLCM	X		Define LNADI-format instructions
.DLMI	X		Define LCALL-format instructions
.DLMO	X		Define LNDO-format instructions
.DLMR	X		Define LPEF-format instructions
.DLMS	X		Define LNDSZ-format instructions
.DLRA	X		Define LLEF-format instructions
.DMR	X		Define JMP-format instructions
.DMRA	X		Define LDA-format instructions
.DTAC	X		Define NOVA-4 LDB-format instructions
.DUNR	X		Define WRTN and other instructions
.DUNS	X		Define DSZTS and other instructions
.DUSR	X		Define a numeric symbol
.DUWR	X		Define WDINC-format instructions
.DUWS	X		Define NFSSS-format instructions
.DWMM	X		Define DERR-format instructions
.DWMR	X		Define WBR-format instructions
.DWMS	X		Define WSKBO-format instructions
.DWXO	X		Define WXOP-format instruction
.DXBA	X		Define XLEFB-format instruction
.DXBR	X		Define XPEFB-format instruction
.DXCM	X		Define XNADI-format instruction
.DXMI	X		Define XCALL-format instruction
.DXMO	X		Define XNDO-format instruction
.DXMR	X		Define XPEF-format instruction
.DXMS	X		Define XNDSZ-format instruction
.DXOP	X		Define XOP-format instruction
.DXRA	X		Define XLEF-format instruction
.XPNG	X		Delete symbols from symbol table

Miscellaneous Pseudo-Ops

A number of additional pseudo-ops do not fit into any previous category. These pseudo-ops are listed in Table 6-13.

Some of them set miscellaneous data in the object file. The .CSIZE pseudo-op sets the size of the unlabelled common area. The .FORCE pseudo-op sets the force-link flag, which is used by LFE and Link. The .REV pseudo-op sets the object's revision number. The .TITLE pseudo-op sets the object's title. And the .TSK pseudo-op sets the maximum number of concurrent tasks.

The .OB pseudo-op sets the name of the object file, overriding the default of the first source file. These pseudo-ops may, in turn, be overridden by the /O= command-line switch. The .RB pseudo-op, which has the same form as .OB, is provided only for compatibility with other Data General assemblers.

The .ENABLE pseudo-op allows user control of certain defaults used by MASM. .ENABLE ABS and .ENABLE PCREL change the default indexing mode used in assembling instructions without an explicit indexing mode. .ENABLE DWORD, .ENABLE SWORD, .ENABLE UWORD and .ENABLE WORD set the size and expected range of values for data items given without a data-formatting pseudo-op.

The .PASS and .SKIP pseudo-ops allow access at assembly time to certain assembler data. .PASS is a value pseudo-op equal to zero on the assembler's first pass, and one on its second pass. The .SKIP pseudo-op returns one if the previous instruction could possibly skip the next data word.

Table 6-13 Miscellaneous Pseudo-Ops

Pseudo-op	Directive	Value	Description
.CSIZ, .CSIZE	X		Set size of unnamed common area
.ENABLE	X		Control defaults
.FORC, .FORCE	X		Set library force-load flag
.OB	X		Name object file
.PASS		X	Return pass number
.RB	X		Name object file
.REV	X		Set object module revision number
.SKIP	X	X	Flag if previous instruction may skip
.TITL, .TITLE	X		Set object module title
.TSK	X		Set maximum concurrent task count

End of Chapter

7

Pseudo-Op Dictionary

This Chapter describes all the AOS/VS Macroassembler 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
- 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
- references for further information about related topics
- the syntax of the pseudo-op as an assembler directive, if applicable. Please note that we use square brackets in a syntax line to set off those arguments or items that are optional.

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

```
01                                          .NREL
02 000000 UC  00000000003                   3
03            00000000004  UC               .LOC .+2    ;Set the location counter equal to
04                                                      ;its current value plus two words.
05                                                      ;Note that the relocation base
06 000004 UC 101771                         LWLDA 0,10 ;Does not change (columns 11-12).
07            00000000010
```

References

"Location Counter" - Chapter 3
"Relocatability" - Chapter 3

Align locations to a memory boundary .ALIGN

Syntax

.ALIGN□**abs-expr**

Purpose

The .ALIGN pseudo-op establishes the word alignment of the next memory word in the current partition. The result of **abs-expr** should be a nonrelocatable expression, n, in the range 0 to 10 (decimal). The next memory word aligns to a memory address which is a multiple of 2^n.

In order to force a particular location in a relocatable partition to be aligned, you must align the entire partition to a 2^n address boundary. AOS/VS MASM automatically takes the maximum of all .ALIGN values for each partition and sets its alignment attribute to that maximum.

Example

```
.
.
.ALIGN  1
2       ; This will be on a double-word boundary
.
.
.ALIGN  10.
2       ; This will be on a page (1024-word) boundary
.
.
```

Number of arguments passed to a macro — .ARGCT

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.

NOTE: *.ARGC is an acceptable abbreviation of this pseudo-op.*

Examples

```
01              00000000000             .NREL  0
02                                      .MACRO ARG  ;Define macro ARG.
03                                      ^1+^2
04                                      (.ARGCT)
05                              %
06                                                  ;Call ARG with 2 arguments
07                                      ARG  4,5    ;(value of .ARGCT is 2).
08 000000 UC    00000000011             4+5
09 000002 UC    00000000002             (.ARGCT)
```

```
01                                      .MACRO ARG      ;Define macro ARG.
02                                      .IFE   .ARGCT   ;If you call ARG with no
03                                      10              ;arguments, assemble the
04                                      .ELSE           ;value 10.  Otherwise,
05                                      ^1              ;assemble the value of
06                                      .ENDC           ;the first argument.
07                              %
08
09                                      ARG             ;Call ARG without arguments.
10              00000000001             .IFE   .ARGCT   ;If you call ARG with no
11 000000       00000000010             10              ;arguments, assemble the
12                                      .ELSE           ;value 10.  Otherwise,
13                                                      ;assemble the value of
14                                      .ENDC           ;the first argument.
15
16                                      ARG    2
17              00000000000             .IFE   .ARGCT   ;If you call ARG with no
18                                      10              ;arguments, assemble the
19                                      .ELSE           ;value 10.  Otherwise,
20 000002       00000000002             2               ;assemble the value of
21                                      .ENDC           ;the first argument.
22
23
```

References

"Macros" - Chapter 5
"Macro-Related Pseudo-Ops" - Chapter 5

Define an accumulating symbol .ASYM

Syntax

.ASYM□**numeric-symbol**

Purpose

This pseudo-op defines a **numeric-symbol** whose value, as an external, will be the sum of the values assigned to it as an accumulating symbol. For this summing to take place, **numeric-symbol** must be declared as a .ASYM and assigned a value. To access its accumulated value, it must be declared as an external (.EXTL) in a separate module. You cannot declare **numeric-symbol** as both a .ASYM and a .EXTL in the same module. Its intermediate values are not accessible. The relocation property of the symbol is not preserved.

Example

```
.TITLE   A
.ASYM    X
X=5
.END


.TITLE   B
.ASYM    X
X=7
.END


.TITLE   C
.EXTLD   X
X                ; X will have value 12. in this module.
.END
```

Reserve a block of memory .BLK

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 nonnegative absolute expression.

The assembler increments the current location counter by **abs-expr** when it encounters .BLK in your source.

Example

```
01              00000000000             .NREL 0
02 000000 UC 122371                     LWSTA 0,ACLOC   ;Store the four accumulators
03              00000000013
04 000003 UC 126371                     LWSTA 1,ACLOC+2 ;in consecutive locations
05              00000000012
06 000006 UC 132371                     LWSTA 2,ACLOC+4 ;starting at address ACLOC.
07              00000000011
08 000011 UC 136371                     LWSTA 3,ACLOC+6
09              00000000010
10
11 000014 UC  00000000010               ACLOC: .BLK 10  ;Save 10 (octal) words of
12                                                      ;memory.
13 000024 UC 101771                     LWLDA  0,150    ;Note that the location
14              00000000150
15                                                      ;counter jumps 10 words.
16
17
```

References

"Absolute Expressions" - Chapter 3
"Assigning Locations" - Chapter 3

Create a data word with call relocation .CALL

Syntax

.CALL□**abs-expr**□**numeric-symbol**

Purpose

The .CALL pseudo-op generates one data word with a value of **abs-expr**, and call relocates it relative to **numeric-symbol**. **Numeric-symbol** must be an external (.EXTN) symbol, and should resolve to a .PENT symbol. .CALL, along with .TARG, generates call and target data words for using the AOS and AOS/VS 16-bit resource manager. The Link utility resolves the contents of call and target words based on the value of **abs-expr** and the program's overlay structure. See the *AOS/VS Link and Link File Editor (LFE) User's Manual* for a detailed discussion of resource calls.

Example

```
        .EXTN   P
        .
        .
        .CALL   4,P     ; ?RCALL resource call word
        .TARG   4,P     ; ?RCALL resource target word
        .
        .
```

Reserve a labeled common area .COMM

Syntax

.COMM□**numeric-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 can access from separately assembled modules in your program.

The assembler assigns the name **numeric-symbol** to this common area. MASM regards **numeric-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 .EXTL to declare **numeric-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

```
.TITLE  A       ;Module A.
.COMM   X,30    ;Reserve a common area named X
                ;of length 30 words.
.COMM   Y,20    ;Common area Y contains 20 words.
        .
        .
        .
.END


.TITLE  B       ;Separately assembled module B.
.COMM   X,30    ;X refers to the same common area as
        .       ;declared in module A.
        .
        .
.END


.TITLE  C       ;Separately assembled module C.
.EXTL   X       ;X is defined in a different module
        .       ;and, in this case, refers to the
        .       ;starting address of the common
        .       ;area declared in module A.
.END
```

References

"Absolute Expressions" – Chapter 3
AOS/VS Link and Link File Editor (LFE) User's Manual – labeled common areas
"Intermodule Communication" – Chapter 6
"Numeric symbols" – Chapter 2

Reserve an unlabeled common area .CSIZE

Syntax

.CSIZE□**abs-expr**

Purpose

The .CSIZE pseudo-op reserves an unlabeled common area for intermodule communication. A common area is a data storage area that you can access from separately assembled modules in your program.

The size of this unlabeled common area is equal to the number of 16-bit words you specify in the **abs-expr** argument. This argument must be a positive absolute expression.

Link assigns the name ?CLOC to the starting address of your unlabeled common area. To reference this common area from a separately assembled module, declare ?CLOC in an .EXTN or .EXTL pseudo-op. The *AOS/VS Link and Link File Editor (LFE) User's Manual* provides more information about ?CLOC and unlabeled common areas.

If you include more than one .CSIZE 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 .CSIZE pseudo-ops, Link uses the largest value.

NOTE: .CSIZ is an acceptable abbreviation of this pseudo-op.

Example

```
.TITLE  A       ;Module A.
.CSIZE   100     ;Reserve an unlabeled common area
        .       ;100 words long.
        .
        .
.END


.TITLE  B       ;Separately assembled module B.
.EXTL   ?CLOC   ;?CLOC is the starting address of
        .       ;the unlabeled common area declared
        .       ;in module A.
        .
.END
```

References

"Absolute Expressions" - Chapter 3
AOS/VS Link and Link File Editor (LFE) User's Manual
- ?CLOC and unlabeled common areas
"Intermodule Communication" - Chapter 6

Define an ALC instruction .DALC

Syntax

.DALC□**numeric-symbol** = instruction or expression

Purpose

The .DALC pseudo-op defines **numeric-symbol** as an ALC-format instruction. When used, an ALC instruction accepts two or three arguments, and several other options, as discussed below.

The syntax of an ALC instruction is as follows:

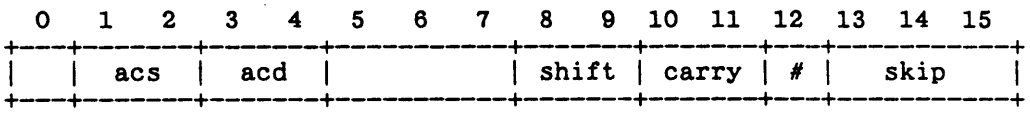
numeric-symbol[<carry>][<shift>][#] <acs> <acd> [<skip>]

Where

<carry>	is either omitted, Z, O or C
<shift>	is either omitted, L, R or S
#	is the ALC instruction no-load indicator
<acs>	is an absolute expression between 0 and 3
<acd>	is an absolute expression between 0 and 3
<skip>	is an absolute expression between 0 and 7

The carry and shift options are available only if the length of the numeric-symbol name is exactly three characters.

MASM assembles an ALC instruction and its arguments into one word in the following format:



AOS/VS MASM includes the following .DALC-defined instructions in its permanent table:

ADC	ADD	AND	COM	INC
MOV	NEG	SUB		

In addition, AOS/VS MASM includes the following mnemonics for the skip option in its permanent table:

SKP	SZC	SNC	SZR	SNR
SEZ	SBN			

Define commercial memory reference instruction .DCMR

Syntax

.DCMR□**numeric-symbol** = instruction or expression

Purpose

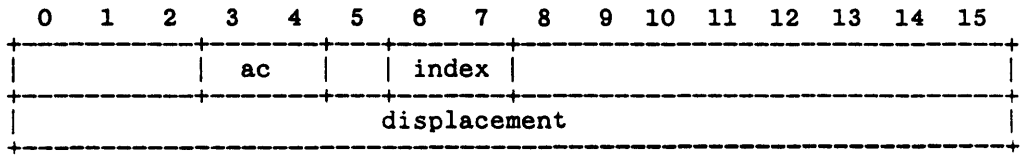
The syntax of a .DCMR instruction is as follows:

numeric-symbol <ac> <displacement> [<index>]

Where

<ac>	is an absolute expression between 0 and 3
<displacement>	is either an unsigned relocatable expression between 0 and 177777 or a signed relocatable expression between -77777 and +77777
<index>	is an absolute expression between 0 and 3

MASM assembles the .DCMR instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DCMR-defined instructions in its permanent table:

ELDB ESTB

Define extended memory reference instruction .DEMR

Syntax

.DEMR□**numeric-symbol** = instruction or expression

Purpose

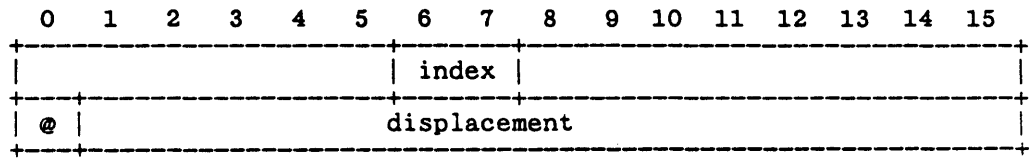
The syntax of a .DEMR instruction is as follows:

numeric-symbol [@]<displacement> [<index>]

Where

@	is the indirection indicator
<displacement>	is either an unsigned relocatable expression between 0 and 77777 or a signed relocatable expression between -37777 and +37777.
<index>	is an absolute expression between 0 and 3

MASM assembles the .DEMR instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DEMR-defined instructions in its permanent table:

EDSZ EISZ EJMP EJSR PSHJ

Define extended memory reference instruction with accumulator .DERA

Syntax

.DERA□**numeric-symbol** = instruction or expression

Purpose

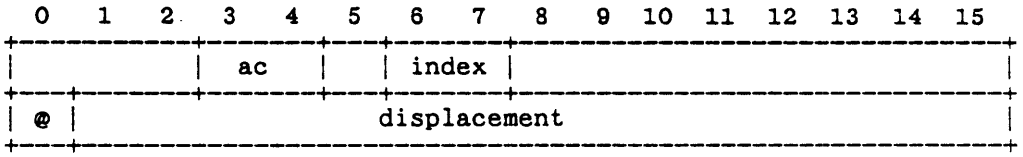
The syntax of a .DERA instruction is as follows:

numeric-symbol <ac> [@]<displacement> [<index>]

Where

<ac>	is an absolute expression between 0 and 3
@	is the indirection indicator
<displacement>	is either an unsigned relocatable expression between 0 and 77777 or a signed relocatable expression between -77777 and +77777.
<index>	is an absolute expression between 0 and 3

MASM assembles the .DERA instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DERA-defined instructions in its permanent table:

DSPA ELDA ELEF ESTA

Define extended user instruction .DEUR

Syntax

.DEUR□**numeric-symbol** = instruction or expression

Purpose

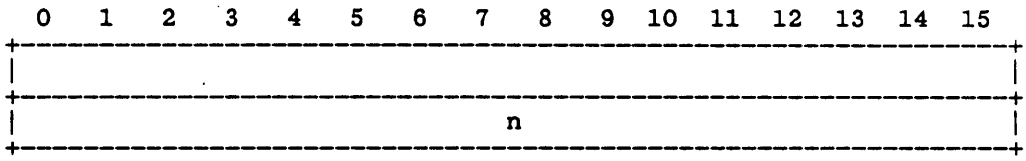
The syntax of a .DEUR instruction is as follows:

numeric-symbol <n>

Where

<n> is an expression between 0 and 177777

MASM assembles the .DEUR instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DEUR-defined instructions in its permanent table:

SAVE	VCT	WSAVR	WSAVS	WSSVR
WSSVS	WWCS	XVCT		

Define floating-point and memory instruction .DFLM

Syntax

.DFLM□**numeric-symbol** = instruction or expression

Purpose

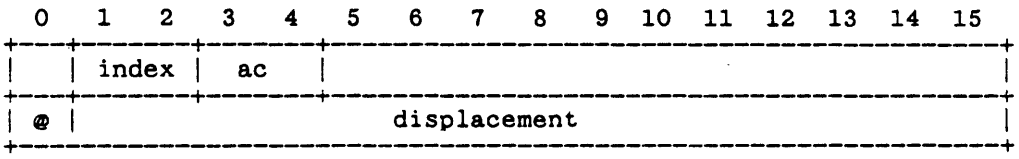
The syntax of a .DFLM instruction is as follows:

numeric-symbol <ac> [@]<displacement> [<index>]

Where

<ac>	is an absolute expression between 0 and 3
@	is the indirection indicator
<displacement>	is either an unsigned relocatable expression between 0 and 77777 or a signed relocatable expression between -37777 and +37777
<index>	is an absolute expression between 0 and 3

MASM assembles the .DFLM instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DFLM-defined instructions in its permanent table:

FAMS	FAMD	FSMS	FSMD	FMMS
FMMD	FDMS	FDMD	FLDS	FLDD
FSTS	FSTD	FFMD	FLMD	

Define floating load status instruction .DFLS

Syntax

.DFLS□**numeric-symbol** = instruction or expression

Purpose

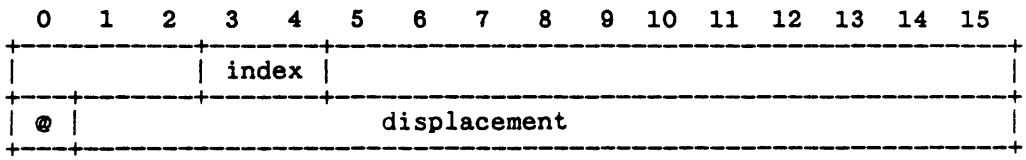
The syntax of a .DFLS instruction is as follows:

numeric-symbol [@]<displacement> [<index>]

Where

@	is the indirection indicator
<displacement>	either an unsigned relocatable expression between 0 and 77777 or a signed relocatable expression between -37777 and +37777
<index>	is an absolute expression between 0 and 3

MASM assembles the .DFLS instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DFLS-defined instructions in its permanent table:

FLST FSST

Define one-accumulator instruction .DIAC

Syntax

.DIAC□**numeric-symbol** = instruction or expression

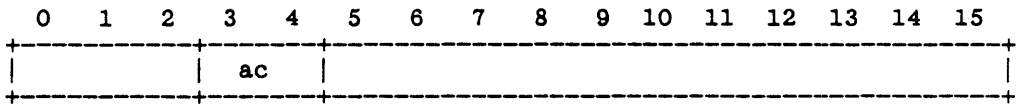
Purpose

The syntax of a .DIAC instruction is as follows:

numeric-symbol <ac>

Where

<ac> is an absolute expression between 0 and 3. MASM assembles the .DIAC instruction and its argument into one word in the following format:



AOS/VS MASM includes the following .DIAC-defined instructions in its permanent table:

CVWN	FAB	FEXP	FHLV	FINT
FNEG	FNOM	FRH	FSCAL	HLV
INTA	LDAFP	LDASB	LDASL	LDASP
LDATS	LDI	MSKO	MSP	READS
STAFP	STASB	STASL	STASP	STATS
STI	WHLV	WLDI	WMOVR	WMSP
WSTI	XCT			

Define one-accumulator instruction with immediate .DICD

Syntax

.DICD□**numeric-symbol** = instruction or expression

Purpose

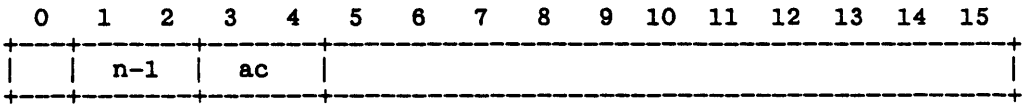
The syntax of a .DICD instruction is as follows:

numeric-symbol <n> <ac>

Where

<n>	is an absolute expression between 1 and 4
<ac>	is an absolute expression between 0 and 3

MASM automatically subtracts 1 from n and assembles the .DICD instruction and its arguments into one word in the following format:



AOS/VS MASM includes the following .DICD-defined instructions in its permanent table:

ADI	DHXL	DHXR	HXL	HXR
NADI	NSBI	SBI	WADI	WLSI
WSBI				

Define immediate and two-accumulator instruction .DIMD

Syntax

.DIMD□**numeric-symbol** = instruction or expression

Purpose

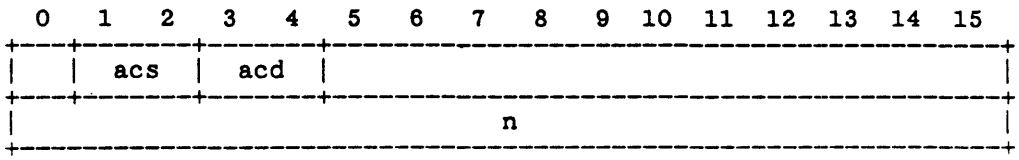
The syntax of a .DIMD instruction is as follows:

numeric-symbol <n> <acs> <acd>

Where

<n>	is an expression between -177777 and +177777
<acs>	is an absolute expression between 0 and 3
<acd>	is an absolute expression between 0 and 3

MASM assembles the .DIMD instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DIMD-defined instruction in its permanent table:

CIOI

Define immediate and one-accumulator instruction .DIMM

Define immediate and one-accumulator instruction which may skip .DIMS

Syntax

.DIMM□**numeric-symbol** = instruction or expression

.DIMS□**numeric-symbol** = instruction or expression

Purpose

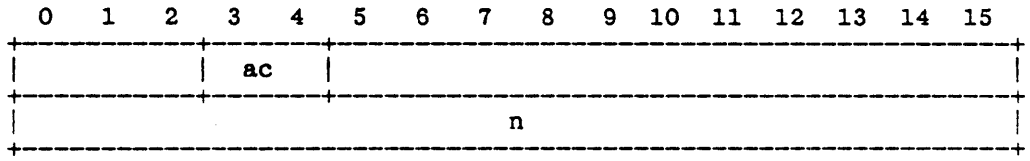
The syntax of a .DIMM or .DIMS instruction is as follows:

numeric-symbol <n> <ac>

Where

<n>	is an expression between -177777 and +177777
<ac>	is an absolute expression between 0 and 3

MASM assembles the .DIMM or .DIMS instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DIMM-defined instructions in its permanent table:

ADDI	ANDI	IORI	NADDI	NLDAI
WASHI	WLSHI	WNADI	XORI	

AOS/VS MASM includes the following .DIMS-defined instructions in its permanent table:

NSALA	NSALM	NSANA	NSANM	WSEQI
WSGTI	WSLEI	WSNEI		

Define I/O instruction .DIO

Syntax

.DIO□**numeric-symbol** = instruction or expression

Purpose

The syntax of a .DIO instruction is as follows:

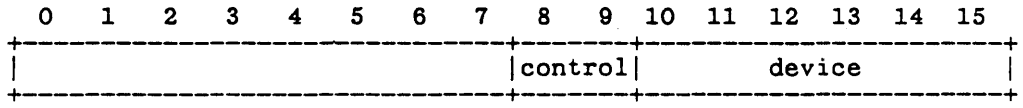
numeric-symbol [<control>] <device>

Where

<control>	is either omitted, S, C or P
<device>	is an absolute expression between 0 and 77

The control option is available only if the length of the name of **numeric-symbol** is exactly three characters.

MASM assembles the .DIO instruction and its arguments into one word in the following format:



AOS/VS MASM includes the following .DIO-defined instructions in its permanent table:

NIO SKPBN SKPBZ SKPDN SKPDZ

AOS/VS MASM also includes mnemonics for the device codes in its permanent table.

Define I/O instruction with accumulator .DIOA

Syntax

.DIOA□**numeric-symbol** = instruction or expression

Purpose

The syntax of a .DIOA instruction is as follows:

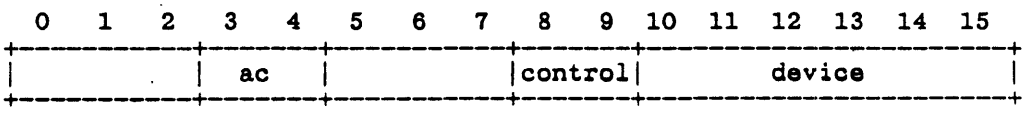
numeric-symbol[<control>] <ac> <device>

Where

<control>	is either omitted, S, C or P
<ac>	is an absolute expression between 0 and 3
<device>	is an absolute expression between 0 and 77

The control option is available only if the length of the name of **numeric-symbol** is exactly three characters.

MASM assembles the .DIOA instruction and its arguments into one word in the following format:



AOS/VS MASM includes the following .DIOA-defined instructions in its permanent table:

DIA DIB DIC DOA DOB
DOC

AOS/VS MASM also includes mnemonics for the device codes in its permanent table.

Define instruction with source and destination .DISD

Define instruction with source and destination which may skip .DISS

Syntax

.DISD□**numeric-symbol** = instruction or expression

.DISS□**numeric-symbol** = instruction or expression

Purpose

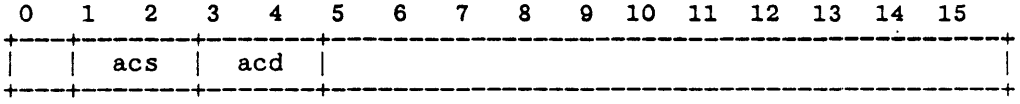
The syntax of a .DISD or .DISS instruction is as follows:

numeric-symbol <acs> <acd>

Where

<acs>	is an absolute expression between 0 and 3
<acd>	is an absolute expression between 0 and 3

MASM assembles the .DISD or .DISS instruction and its arguments into one word in the following format:



AOS/VS MASM includes the following .DISD-defined instructions in its permanent table:

ANC	BTO	BTZ	CIO	CLM
COB	DAD	DLSH	DSB	FAS
FAD	FSS	FSD	FMS	FMD
FDS	FDD	FLAS	FFAS	FCMP
FMOV	FRDS	IOR	LDB	LOB
LRB	LSH	NADD	NDIV	NMUL
NNEG	NSUB	PIO	POP	PSH
SEX	STB	SYC	WADC	WADD
WANC	WAND	WASH	WBTO	WBTZ
WCLM	WCOB	WCOM	WDIV	WFLAD
WFFAD	WINC	WIOR	WLDB	WLOB
WLRB	WLSH	WMOV	WMUL	WNEG
WPOP	WPSH	WSTB	WSUB	WXCH
WXOR	XCH	XOR	ZEX	

.DISD, .DISS (continued)

AOS/VS MASM includes the following .DISS-defined instructions in its permanent table:

SGE	SGT	SNB	SZB	SZBO
USEQ	USGE	USGT	USLE	USLT
USNE	WSEQ	WSNE	WSLE	WSLT
WSGE	WSGT	WSNB	WSZB	WSZBO
WUSGE	WUSGT			

Define wide immediate with one accumulator instruction .DIWM

Define wide immediate with one accumulator instruction which may skip .DIWS

Syntax

.DIWM□**numeric-symbol** = instruction or expression

.DIWS□**numeric-symbol** = instruction or expression

Purpose

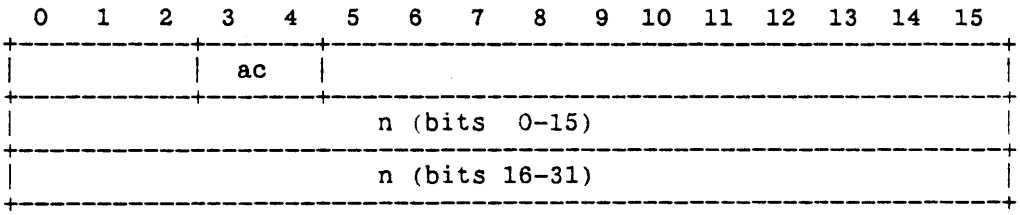
The syntax of a .DIWM or .DIWS instruction is as follows:

numeric-symbol <n> <ac>

Where

<n>	is an expression between -37777777777 and +37777777777
<ac>	is an absolute expression between 0 and 3

MASM assembles the .DIWM or .DIWS instruction and its arguments into three words in the following format:



AOS/VS MASM includes the following .DIWM-defined instructions in its permanent table:

WADDI WANDI WIORI WLDAI WXORI

AOS/VS MASM includes the following .DIWS-defined instructions in its permanent table:

WSALA WSALM WSANA WSANM WUGTI
WULEI

Define a 32-bit byte reference instruction with accumulator .DLBA

Syntax

.DLBA□**numeric-symbol** = instruction or expression

Purpose

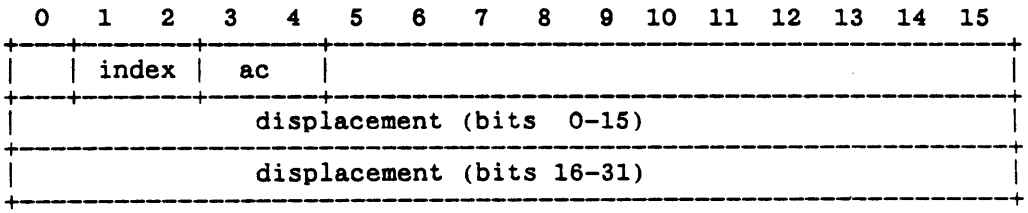
The syntax of a .DLBA instruction is as follows:

numeric-symbol <ac> <displacement> [<index>]

Where

<ac>	is an absolute expression between 0 and 3
<displacement>	is either an unsigned expression between 0 and 37777777777 or a signed expression between -17777777777 and +17777777777
<index>	is an absolute expression between 0 and 3

MASM assembles the .DLBA instruction and its arguments into three words in the following format:



AOS/VS MASM includes the following .DLBA-defined instructions in its permanent table:

LLDB LLEFB LSTB

Define a 32-bit byte reference instruction .DLBR

Syntax

.DLBR□**numeric-symbol** = instruction or expression

Purpose

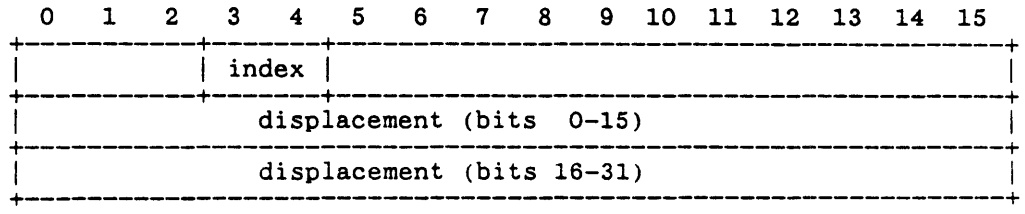
The syntax of a .DLBR instruction is as follows:

numeric-symbol <displacement> [<index>]

Where

<displacement>	is either an unsigned expression between 0 and 37777777777 or a signed expression between -17777777777 and +17777777777
<index>	is an absolute expression between 0 and 3

MASM assembles the .DLBR instruction and its arguments into three words in the following format:



AOS/VS MASM includes the following .DLBR-defined instruction in its permanent table:

LPEFB

Define a 32-bit memory reference with immediate .DLCM

Syntax

.DLCM□**numeric-symbol** = instruction or expression

Purpose

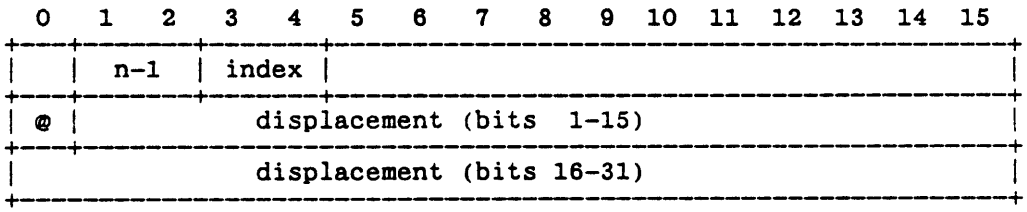
The syntax of a .DLCM instruction is as follows:

numeric-symbol <n> [@]<displacement> [<index>]

Where

<n>	is an absolute expression between 1 and 4
@	is the indirection indicator
<displacement>	is either an unsigned expression between 0 and 17777777777 or a signed expression between -7777777777 and +7777777777
<index>	is an absolute expression between 0 and 3

MASM automatically subtracts 1 from n and assembles the .DLCM instruction and its arguments into three words in the following format:



AOS/VS MASM includes the following .DLCM-defined instructions in its permanent table:

LNADI LNSBI LWADI LWSBI

Define a 32-bit memory reference with immediate .DLMI

Syntax

.DLMI□**numeric-symbol** = instruction or expression

Purpose

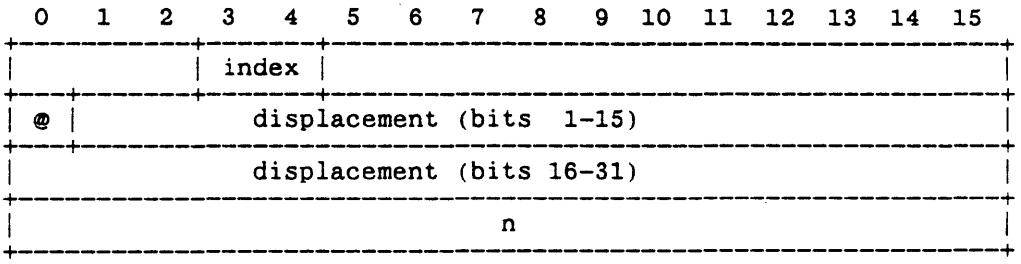
The syntax of a .DLMI instruction is as follows:

numeric-symbol [@]<displacement> [<index> [<n>]]

Where

@	is the indirection indicator
<displacement>	is either an unsigned expression between 0 and 17777777777 or a signed expression between -7777777777 and +7777777777
<index>	is an absolute expression between 0 and 3
<n>	is an absolute expression between 0 and 177777

MASM assembles the .DLMI instruction and its arguments into four words in the following format:



AOS/VS MASM includes the following .DLMI-defined instruction in its permanent table:

LCALL

Define a 32-bit memory reference with accumulator and offset .DLMO

Syntax

.DLMO□**numeric-symbol** = instruction or expression

Purpose

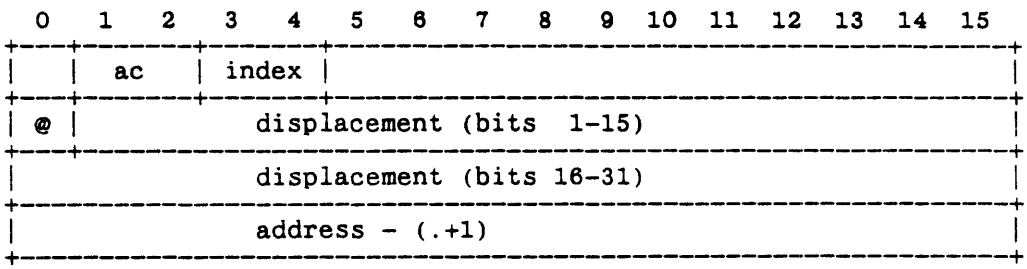
The syntax of a .DLMO instruction is as follows:

numeric-symbol <ac> <address> [@]<displacement> [<index>]

Where

<ac>	is an absolute expression between 0 and 3
<address>	is an address in the current partition
@	is the indirection indicator
<displacement>	is either an unsigned expression between 0 and 17777777777 or a signed expression between -7777777777 and +7777777777
<index>	is an absolute expression between 0 and 3

MASM subtracts .+1 from address and assembles the .DLMO instruction and its arguments into four words in the following format:



AOS/VS MASM includes the following .DLMO-defined instructions in its permanent table:

LNDO LWDO

Define a 32-bit memory reference .DLMR

Define a 32-bit memory reference which may skip .DLMS

Syntax

.DLMR□**numeric-symbol** = instruction or expression

.DLMS□**numeric-symbol** = instruction or expression

Purpose

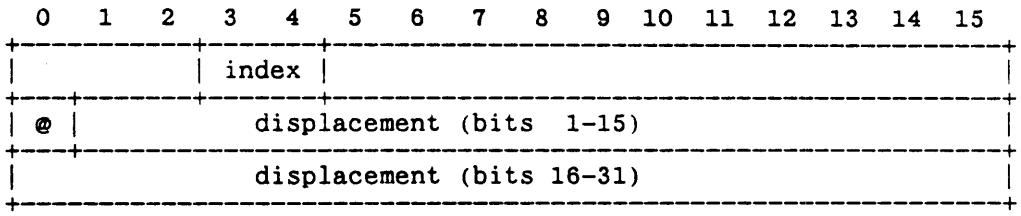
The syntax of a .DLMR or .DLMS instruction is as follows:

numeric-symbol [@]<displacement> [<index>]

Where

@	is the indirection indicator
<displacement>	is either an unsigned expression between 0 and 17777777777 or a signed expression between -7777777777 and +7777777777
<index>	is an absolute expression between 0 and 3

MASM assembles the .DLMR or .DLMS instruction and its arguments into three words in the following format:



AOS/VS MASM includes the following .DLMR-defined instructions in its permanent table:

LFLST LFSST LJMP LJSR LPEF
LPSHJ

AOS/VS MASM includes the following .DLMS-defined instructions in its permanent table:

LNDSZ LNISZ LWDSZ LWISZ

Define a 32-bit memory reference with accumulator .DLRA

Syntax

.DLRA□**numeric-symbol** = instruction or expression

Purpose

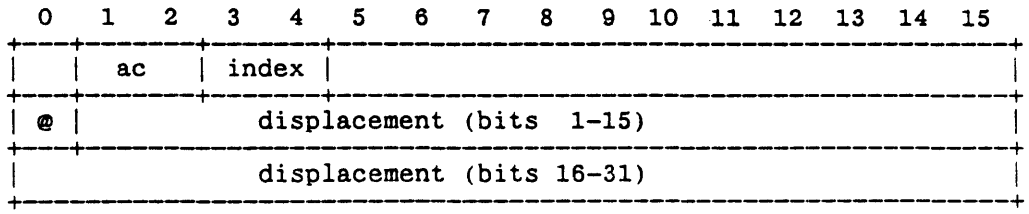
The syntax of a .DLRA instruction is as follows:

numeric-symbol <ac> [@]<displacement> [<index>]

Where

<ac>	is an absolute expression between 0 and 3
@	is the indirection indicator
<displacement>	is either an unsigned expression between 0 and 17777777777 or a signed expression between -7777777777 and +7777777777
<index>	is an absolute expression between 0 and 3

MASM assembles the .DLRA instruction and its arguments into three words in the following format:



AOS/VS MASM includes the following .DLRA-defined instructions in its permanent table:

LDSP	LFLDS	LFLDD	LFSTS	LFSTD
LFAMS	LFAMD	LFSMS	LFSMD	LFMMS
LFMMD	LFDMS	LFDMD	LLEF	LNLDA
LNSTA	LNADD	LNSUB	LNMUL	LNDIV
LWLDA	LWSTA	LWADD	LWSUB	LWMUL
LWDIV				

Define an 8-bit memory reference .DMR

Syntax

.DMR□**numeric-symbol** = instruction or expression

Purpose

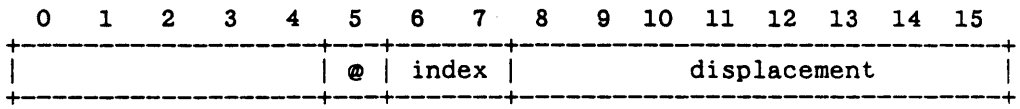
The syntax of a .DMR instruction is as follows:

numeric-symbol [@]<displacement> [<index>]

Where

@	is the indirection indicator
<displacement>	is either an unsigned expression between 0 and 377 or a signed expression between −177 and +177
<index>	is an absolute expression between 0 and 3

MASM assembles the .DMR instruction and its arguments into one word in the following format:



AOS/VS MASM includes the following .DMR-defined instructions in its permanent table:

DSZ ISZ JMP JSR

Define an 8-bit memory reference with accumulator .DMRA

Syntax

.DMRA□**numeric-symbol** = instruction or expression

Purpose

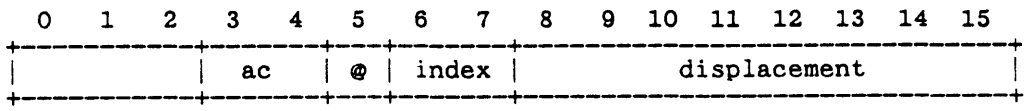
The syntax of a .DMRA instruction is as follows:

numeric-symbol <ac> [@]<displacement> [<index>]

Where

<ac>	is an absolute expression between 0 and 3
@	is the indirection indicator
<displacement>	is either an unsigned expression between 0 and 377 or a signed expression between -177 and +177
<index>	is an absolute expression between 0 and 3

MASM assembles the .DMRA instruction and its arguments into one word in the following format:



AOS/VS MASM includes the following .DMRA-defined instructions in its permanent table:

LDA LEF STA

Assemble source lines repetitively .DO

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 the .DO loop with the .ENDC pseudo-op. 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 can 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 can 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.

Examples

Source code for the first example:

```
        .DO     3       ;Assemble the following code 3 times.
        10
        20
        .ENDC           ;End of .DO loop.
```

Assembly listing for this code:

```
01                00000000003           .DO     3       ;Assemble the following code 3 times.
02 000000         00000000010           10
03 000002         00000000020           20
04                                      .ENDC           ;End of .DO loop.
05 000004         00000000010           10
06 000006         00000000020           20
07                                      .ENDC           ;End of .DO loop.
08 000010         00000000010           10
09 000012         00000000020           20
10                                      .ENDC           ;End of .DO loop.
```

.DO (continued)

The second example shows how to use the .DO pseudo-op to perform conditional assembly:

```
A=3
.DO     A==3    ;Assemble the following code once
3               ;if the value of A equals 3.  Otherwise,
.ENDC           ;do not assemble the code at all.
```

References

"Absolute Expressions" - Chapter 3
"Loops and Conditionals in Macros" - Chapter 5
"Repetitive and Conditional Assembly" - Chapter 6

Define a two-accumulator instruction .DTAC

Syntax

.DTAC□**numeric-symbol** = instruction or expression

Purpose

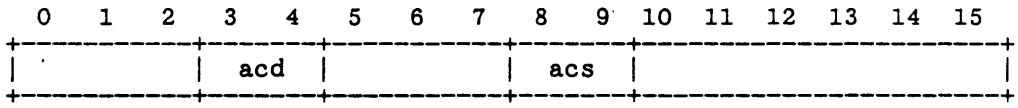
The syntax of a .DTAC instruction is as follows:

numeric-symbol <acs> <acd>

Where

<acs>	is an absolute expression between 0 and 3
<acd>	is an absolute expression between 0 and 3

MASM assembles the .DTAC instruction and its arguments into one word in the following format:



AOS/VS MASM does not include any .DTAC-defined instructions. This instruction format is required only for the LDB and STB instructions on the Data General NOVA4 and microNova machines.

Define a 1-word instruction with no arguments .DUNR

Define a 1-word instruction with no arguments which may skip .DUNS

Syntax

.DUNR□**numeric-symbol** = instruction or expression

.DUNS□**numeric-symbol** = instruction or expression

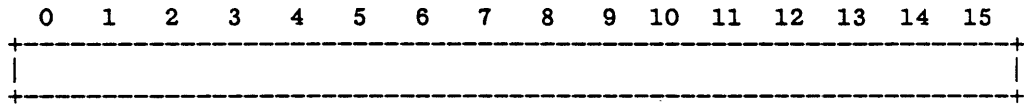
Purpose

The syntax of a .DUNR or .DUNS instruction is as follows:

numeric-symbol

.DUNR and .DUNS instructions do not accept arguments.

MASM assembles the .DUNR or .DUNS instruction into one word in the following format:



AOS/VS MASM includes the following .DUNR-defined instructions in its permanent table:

BAM	BKPT	BLM	CMV	CMP
CMT	CRYTC	CRYTO	CRYTZ	CTR
DIV	DIVS	DIVX	ECLID	EDIT
FCLE	FNS	FTE	FTD	FPSH
FPOP	FPLYD	FPLYS	FSIND	FSINS
FCOSS	FCOSD	FSQRD	FSQRS	FLOGD
FLOGS	FEXPD	FEXPS	FXTD	FXTE
HALT	INTEN	INTDS	IORST	LCPID
LCS	LCSF	LDIX	LMP	LMRF
LPSR	LSBRA	LSBRS	LSN	MUL
MULS	NCLID	ORFB	PATU	PBX
PRTSEL	PSHR	POPB	POPJ	RRFB
RSTR	RTN	SAVZ	SCL	SMRF
SPSR	SSPT	STIX	SVC	WBLM
WCMP	WCMT	WCMV	WCST	WCTR
WDIVS	WDPOP	WEDIT	WFPOP	WFPSH
WLDIX	WLMP	WLSN	WMULS	WPOPB
WPOPJ	WRSTR	WRTN	WSTIX	

.DUNR, .DUNS (continued)

AOS/VS MASM includes the following .DUNS-defined instructions in its permanent table:

DEQUE	DSZTS	ENQH	ENQT	FSA
FSEQ	FSNE	FSLT	FSGE	FSLE
FSGT	FSNM	FSND	FSNU	FSNUD
FSNO	FSNOD	FSNUO	FSNER	ISZTS
LPHY	LPTE	SNOVR	SPTE	VBP
VWP	WMESS			

Define a numeric symbol .DUSR

Syntax

.DUSR□**numeric-symbol** = { **integer** | **symbol** | **expression** | **instruction** }

Purpose

The .DUSR pseudo-op defines **numeric-symbol** as having the value of **integer**, **symbol**, **expression**, or **instruction**.

integer	can be any integer value, but can not be a floating-point constant.
symbol	can be any numeric symbol, instruction symbol, or permanent value symbol (a pseudo-op that represents an internal assembler variable).
expression	can be any legal Macroassembler expression.
instruction	can be any legal MV/Family 32-Bit assembly language instruction. If you supply an instruction, MASM computes the assembled value of that instruction and assigns it to **numeric-symbol**. MASM pads or truncates the instruction's value to produce a double precision (32-bit) integer, if necessary. Refer to "Assignments" in Chapter 2 for more information about using instructions in assignments.

Once defined, you can use **numeric-symbol** anywhere you would use a double precision (32-bit) operand. In addition, you can change the value of **numeric-symbol** at any time without using .XPNG or .DUSR.

The above description shows 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 7-1 assign equivalent values to the symbols A, B, C, and D.

Table 7-1 .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

The only difference between using a simple assignment statement and using the .DUSR pseudo-op concerns multiple definitions of symbols. If you assign a symbol two or more different values with simple assignment statements, no error occurs. However, if you set the

.DUSR (continued)

/MULTIPLE command-line switch , and assign a symbol two or more different values with .DUSR assignments, MASM reports a multiple symbol definition.

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.

Example

```
.DUSR  Z =5    ;Define Z as an assembly-time constant.
```

References

"Expressions" - Chapter 2
"Symbols" - Chapter 2
"Numeric Symbols" - Chapter 2

Define a 2-word instruction with no arguments .DUWR

Define a 2-word instruction with no arguments which may skip .DUWS

Syntax

.DUWR□**numeric-symbol** = instruction or expression

.DUWS□**numeric-symbol** = instruction or expression

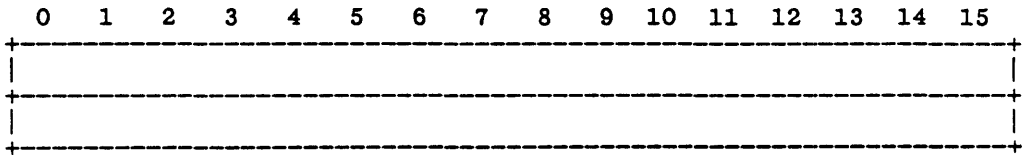
Purpose

The syntax of a .DUWR or .DUWS instruction is as follows:

numeric-symbol

.DUWR and .DUWS instructions do not accept arguments.

MASM assembles the .DUWR or .DUWS instruction into two words in the following format:



AOS/VS MASM includes the following .DUWR-defined instructions in its permanent table:

JPID	JPLOAD	JPFLOAD	WDCMP	WDDEC
WDINC	WDMOV			

AOS/VS MASM includes the following .DUWS-defined instructions in its permanent table:

CINTR	IMODE	JPFLUSH	JPLCS	JPSTART
JPSTATUS	JPSTOP	NFSSS	NBSSS	NFSSC
NBSSC	NFSAS	NBSAS	NFSAC	NBSAC
NFSE	NBSE	NFSGE	NBSGE	NFSLE
NBSLE	NFSNE	NBSNE	WFSSS	WBSSS
WFSSC	WBSSC	WFSAS	WBSAS	WFSAC
WBSAC	WFSE	WBSE	WFSGE	WBSGE
WFSLE	WBSLE	WFSNE	WBSNE	

Define instruction with 5-bit immediate .DWMM

Define instruction with 5-bit immediate which may skip .DWMS

Syntax

.DWMM□**numeric-symbol** = instruction or expression

.DWMS□**numeric-symbol** = instruction or expression

Purpose

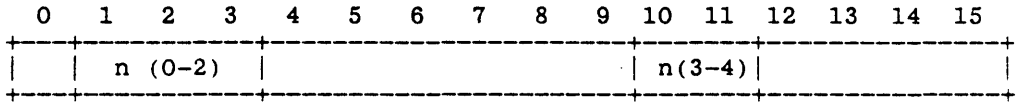
The syntax of a .DWMM or .DWMS instruction is as follows:

numeric-symbol <n>

Where

<n> is an absolute expression between 0 and 37

MASM assembles the .DWMM or .DWMS instruction and its argument into one word in the following format:



AOS/VS MASM includes the following .DWMM-defined instruction in its permanent table:

DERR

AOS/VS MASM includes the following .DWMS-defined instructions in its permanent table:

WSKBO WSKBZ

Define instruction with 8-bit immediate .DWMR

Syntax

.DWMR□**numeric-symbol** = instruction or expression

Purpose

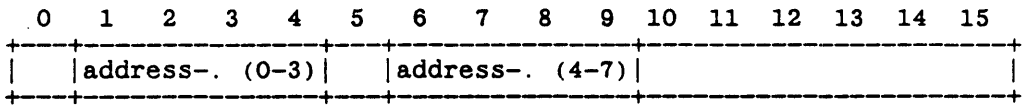
The syntax of a .DWMR instruction is as follows:

numeric-symbol <address>

Where

<address> is an address in the current partition

MASM subtracts . (location counter value) from address and assembles the .DWMR instruction and its argument into one word in the following format:



AOS/VS MASM includes the following .DWMR-defined instruction in its permanent table:

WBR

Store expression in two words .DWORD

Syntax

.DWORD□**expr**

Purpose

The .DWORD pseudo-op directs the assembler to generate a two-word (32-bit) integer with the value and relocation property of **expr**. You can supply any legal assembler expression to .DWORD as an argument.

By default, MASM generates two words of memory for each data placement expression in your source module. Therefore, you need not use this pseudo-op unless you alter the global data placement mode (set with .ENABLE).

Note that the .DWORD pseudo-op does not alter the global data placement mode.

Example

```
01                                      .ENABLE WORD    ;Enable single word data
02 000000    000001                     1               ;placement (global mode).
03 000001    000002                     2
04 000002     00000000004               .DWORD 4,5      ;Store these values in two
05            00000000005
06                                                      ;words each.
07 000006    000006                     6               ;Global mode remains set to
08 000007    000007                     7               ;single word data placement.
```

References

"Data Placement" - Chapter 6
"Expressions" - Chapter 2

Define two-word extended opcode instruction .DWXO

Syntax

.DWXO□**numeric-symbol** = instruction or expression

Purpose

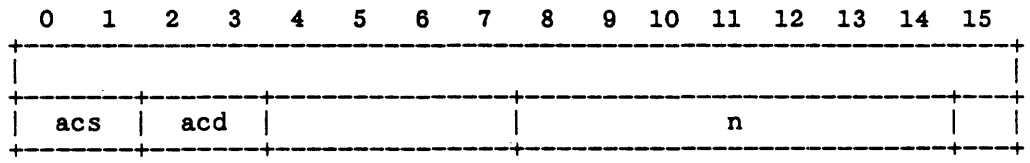
The syntax of a .DWXO instruction is as follows:

numeric-symbol <acs> <acd> <n>

Where

<acs>	is an absolute expression between 0 and 3
<acd>	is an absolute expression between 0 and 3
<n>	is an absolute expression between 0 and 177

MASM assembles the .DWXO instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DWXO-defined instruction in its permanent table:

WXOP

Define 16-bit byte reference with accumulator .DXBA

Syntax

.DXBA□**numeric-symbol** = instruction or expression

Purpose

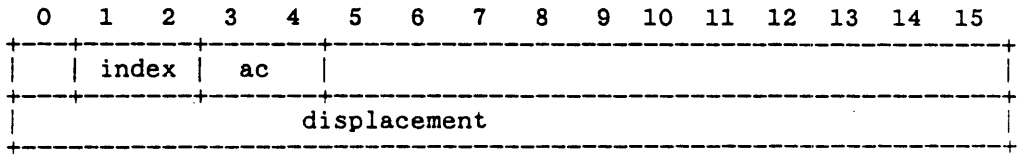
The syntax of a .DXBA instruction is as follows:

numeric-symbol <ac> <displacement> [<index>]

Where

<ac>	is an absolute expression between 0 and 3
<displacement>	is either an unsigned expression between 0 and 177777 or a signed expression between -77777 and +7777777
<index>	is an absolute expression between 0 and 3

MASM assembles the .DXBA instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DXBA-defined instructions in its permanent table:

XLDB XLEFB XSTB

Define 16-bit byte reference with accumulator .DXBR

Syntax

.DXBR□**numeric-symbol** = instruction or expression

Purpose

The syntax of a .DXBR instruction is as follows:

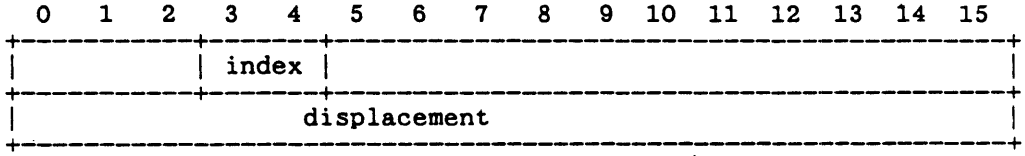
. **numeric-symbol** <displacement> [<index>]

Where

<displacement> is either an unsigned expression between 0 and 177777 or a signed expression between -77777 and +77777

<index> is an absolute expression between 0 and 3

MASM assembles the .DXBR instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DXBR-defined instruction in its permanent table:

XPEFB

Define 16-bit memory reference with immediate .DXCM

Syntax

.DXCM□**numeric-symbol** = instruction or expression

Purpose

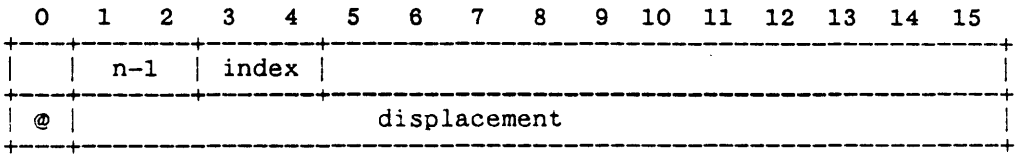
The syntax of a .DXCM instruction is as follows:

numeric-symbol <n> [@]<displacement> [<index>]

Where

<n>	is an absolute expression between 1 and 4
@	is the indirection indicator
<displacement>	is either an unsigned expression between 0 and 77777 or a signed expression between -37777 and +37777
<index>	is an absolute expression between 0 and 3

MASM subtracts 1 from <n> assembles the .DXCM instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DXCM-defined instructions in its permanent table:

XNADI XNSBI XWADI XWSBI

Define 16-bit memory reference with immediate .DXMI

Syntax

.DXMI□**numeric-symbol** = instruction or expression

Purpose

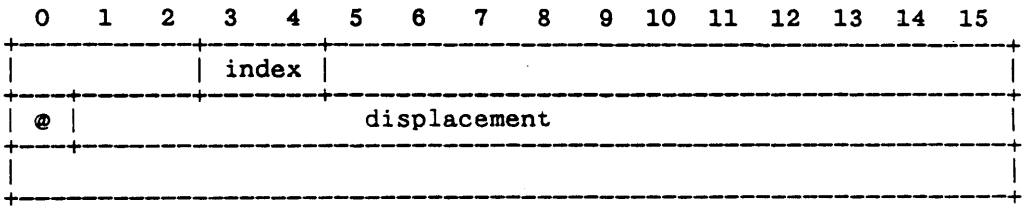
The syntax of a .DXMI instruction is as follows:

numeric-symbol [@]<displacement> [<index> [<n>]]

Where

@	is the indirection indicator
<displacement>	is either an unsigned expression between 0 and 77777 or a signed expression between -37777 and +37777
<index>	is an absolute expression between 0 and 3
<n>	is an absolute expression between 0 and 177777

MASM assembles the .DXMI instruction and its arguments into three words in the following format:



AOS/VS MASM includes the following .DXMI-defined instruction in its permanent table:

XCALL

Define 16-bit memory reference with accumulator and offset .DXMO

Syntax

.DXMO□**numeric-symbol** = instruction or expression

Purpose

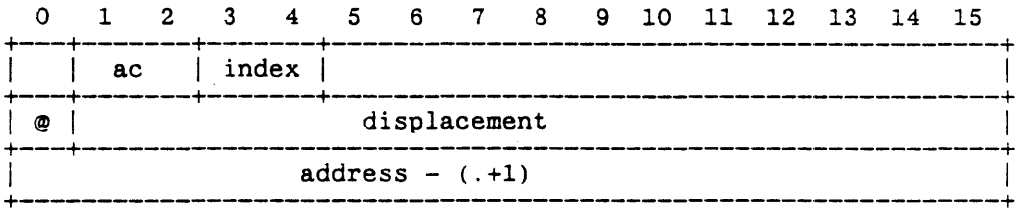
The syntax of a .DXMO instruction is as follows:

numeric-symbol <ac> <address> [@]<displacement> [<index>]

Where

<ac>	is an absolute expression between 0 and 3
<address>	is an address in the current partition
@	is the indirection indicator
<displacement>	is either an unsigned expression between 0 and 77777 or a signed expression between -37777 and +37777
<index>	is an absolute expression between 0 and 3

MASM subtracts .+1 from address and assembles the .DXMO instruction and its arguments into three words in the following format:



AOS/VS MASM includes the following .DXMO-defined instructions in its permanent table:

XNDO XWDO

Define 16-bit memory reference .DXMR

Define 16-bit memory reference which may skip .DXMS

Syntax

.DXMR□**numeric-symbol** = instruction or expression

.DXMS□**numeric-symbol** = instruction or expression

Purpose

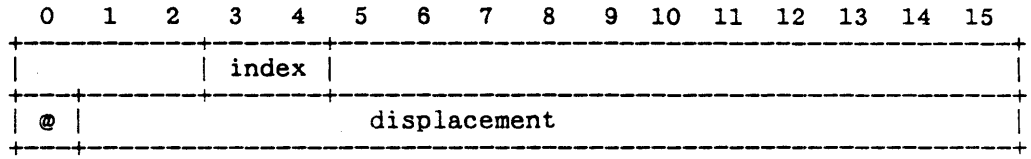
The syntax of a .DXMR or .DXMS instruction is as follows:

numeric-symbol [@]<displacement> [<index>]

Where

@	is the indirection indicator
<displacement>	is either an unsigned expression between 0 and 77777 or a signed expression between -37777 and +37777
<index>	is an absolute expression between 0 and 3

MASM assembles the .DXMR or .DXMS instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DXMR-defined instructions in its permanent table:

XJMP XJSR XPEF XPSHJ

AOS/VS MASM includes the following .DXMS-defined instructions in its permanent table:

XNDSZ XNISZ XWDSZ XWISZ

Define instruction with two accumulators and extended opcode .DXOP

Syntax

.DXOP□**numeric-symbol** = instruction or expression

Purpose

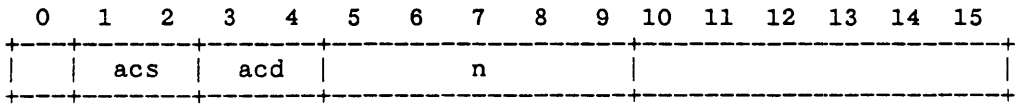
The syntax of a .DXOP instruction is as follows:

numeric-symbol <acs> <acd> <n>

Where

<acs>	is an absolute expression between 0 and 3
<acd>	is an absolute expression between 0 and 3
<n>	is an absolute expression between 0 and 37

MASM assembles the .DXOP instruction and its arguments into one word in the following format:



AOS/VS MASM includes the following .DXOP-defined instructions in its permanent table:

XOP XOP1

Define 16-bit memory reference .DXRA

Syntax

.DXRA□**numeric-symbol** = instruction or expression

Purpose

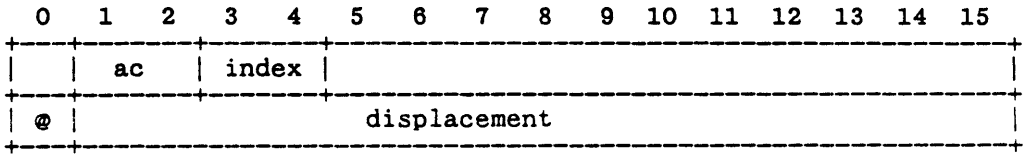
The syntax of a .DXRA instruction is as follows:

numeric-symbol <ac> [@]<displacement> [<index>]

Where

<ac>	is an absolute expression between 0 and 3
@	is the indirection indicator
<displacement>	is either an unsigned expression between 0 and 77777 or a signed expression between -37777 and +37777
<index>	is an absolute expression between 0 and 3

MASM assembles the .DXRA instruction and its arguments into two words in the following format:



AOS/VS MASM includes the following .DXRA-defined instructions in its permanent table:

XFAMD	XFAMS	XFDMD	XFDMS	XFLDD
XFLDS	XFMMD	XFMMS	XFSMD	XFSMS
XFSTD	XFSTS	XLEF	XNADD	XNDIV
XNLDA	XNMUL	XNSTA	XNSUB	XWADD
XWDIV	XWLDA	XWMUL	XWSTA	XWSUB

Begin a new listing page .EJECT

Syntax

.EJECT

Purpose

The .EJECT pseudo-op directs the assembler to begin a new page in the assembly listing output (after listing the .EJECT source statement).

NOTE: *.EJEC is an acceptable abbreviation of this pseudo-op.*

Example

The source code for this example is:

```
          .
          .
          .
MOV       0,1
.EJECT              ;Start a new listing page.
LWLDA     2,0,1
          .
          .
          .
```

The assembly listing for this code is

```
SOURCE: .MAIN               MASM 06.00.00.00       18-JAN-87 16:21:06  PAGE    1


                                                    .
                                                    .
                                                    .
23 000004    105000                                 MOV      0,1
24                                                  .EJECT             ;Start a new listing page.




SOURCE: .MAIN               MASM 06.00.00.00       18-JAN-87 16:21:06  PAGE    2

01 000005    131771                                 LWLDA    2,0,1
02            00000000000
                                                    .
                                                    .
                                                    .
```

Reference

"Assembly Listing" - Chapter 4

Define alternative source code for conditional assembly — .ELSE

Syntax

.ELSE

Purpose

The .ELSE pseudo-op defines source code for MASM to assemble if a conditional expression is false. You must use .ELSE in conjunction with one of the .IF pseudo-ops.

Use the .ENDC pseudo-op to terminate the conditional assembly source lines.

The conditional portion of your source module has the following format:

```
.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.
.ELSE
.                ;Assemble these source lines if abs-expr
.                ;does not satisfy the .IFx condition.
.
.ENDC            ;Terminate conditional portion of source.
```

Example

```
01              00000000003             A=3
02              00000000000             .IFE    A
03                                      100             ;Assembled if A equals 0.
04                                      .ELSE
05 000000       00000000200             200             ;Assembled if A does not equal 0.
06                                      .ENDC
```

References

.IF pseudo-op description - Chapter 7
"Loops and Conditions in Macros" - Chapter 5
"Repetitive and Conditional Assembly"- Chapter 6

Set global data parameters .ENABLE

Syntax

.ENABLE□ { WORD / SWORD / UWORD / DWORD } □ { PCREL / ABS }

Purpose

The .ENABLE pseudo-op allows you to specify a global data placement mode for your source module. It also allows you to specify how the Macroassembler should resolve addresses in memory reference instructions (MRIs).

In the first argument, you direct the assembler either to store data in double words (32-bit storage), or to truncate data values and store them in single words (16-bit storage). In the latter case, you can have the assembler check the high-order bits before truncating them.

The following descriptions summarize the values you can place in the first argument of .ENABLE:

DWORD	Store the results of expressions in double words (32 bits each). This is the default global data placement mode.
WORD	Truncate the results of expressions and store them in single words (16 bits each). The assembler does not return an error if any of the high-order (most significant) 16 bits of the result are set.
SWORD	Truncate the results of expressions, and store them in single words (16 bits each). The assembler returns an error if the high-order 17 bits are not all zeros or ones. That is, the assembler returns an error if each bit in the first word is not equal to the sign bit of the second word.
UWORD	Truncate the results of expressions and store them in single words (16 bits each). The assembler returns an error if any of the high-order 16 bits are set.

Note that the global data placement mode applies only to expressions that appear as data statements. The assembler does not use the global mode to evaluate expressions that appear in other types of statements.

You can override the global data placement mode at any time by issuing one of the following pseudo-ops: .DWORD, .WORD, .SWORD, .UWORD. "Data Placement" in Chapter 6 contains more information about these four pseudo-ops.

The second argument to .ENABLE tells the assembler how to resolve addresses that you use in memory reference (MRI) instructions. When you pass an address in the displacement field of a memory reference instruction, the assembler can resolve that address either with or without reference to the program counter (PC relative or absolute addressing).

.ENABLE (continued)

In PC relative addressing, MASM calculates the address as an offset from the location of the memory reference instruction (the value of the program counter). Thus, MASM passes Link the relocation bases of both the address and the current partition (the partition containing the memory reference instruction).

In absolute addressing, MASM calculates the referenced address without regard to the location of the memory reference instruction. In this case, MASM passes Link the relocation base of the address only; not the base of the memory reference instruction.

To indicate one of these addressing modes, pass either PCREL or ABS as the second argument to .ENABLE. The following descriptions summarize their meanings:

PCREL	MASM calculates each address you supply to a memory reference instruction as an offset from the location of that instruction. That is, MASM resolves the address relative to the current program counter. PCREL is the default value for MRI address resolution.
ABS	MASM calculates each address you supply to a memory reference instruction without regard for the value of the program counter.

Note that the assembler does *not* use the PCREL or ABS declaration when you explicitly indicate a value for the addressing index (mode) in the memory reference instruction.

"Resolving Locations in MRI Instructions" in Chapter 3 provides more information about absolute and PC relative addressing.

In general, you want 32-bit data placement and PC relative external references in memory reference instructions. Thus, DWORD and PCREL are the global settings at the beginning of each assembly.

When you issue the .ENABLE pseudo-op, you need not supply an argument for both global settings; only the one(s) you want to modify. If you supply only one argument, MASM sets that global mode accordingly but does not alter the other setting. For example:

```
.ENABLE          ABS
```

This statement sets the MRI address resolution mode to ABS but does not alter the global data placement mode.

Examples

The first example illustrates the use of .ENABLE to alter data storage placement.

```
01 000000    00000000001          1              ;By default, MASM stores data in
02 000002    00000000002          2              ;double words (DWORD mode).  Set
03                                .ENABLE WORD   ;global data placement mode
04 000004    000003               3              ;to single word.
05 000005    000004               4
06 000006    00000000005          .DWORD  5      ;Store the value 5 in two words.
07 000010    000006               6              ;Global mode remains set to single
08                                               ;word
```

.ENABLE (continued)

The second example shows how .ENABLE can direct the assembler to evaluate external references in memory reference instructions with or without respect to the program counter.

```
01 000000    101771                    LWLDA   0,A      ;MASM resolves address A relative
02            00000000011
03                                                      ;to the program counter, by default.
04                                      .ENABLE ABS     ;Set the global mode to absolute
05 000003    105771                    LWLDA   1,A      ;addressing.  MASM now resolves A
06            00000000011
07                                                      ;without regard to the value
08                                                      ;of the program counter.
09 000006    131771                    LWLDA   2,2,1    ;Since this MRI instruction
10            00000000002
11                                                      ;explicitly includes an addressing
12                                                      ;index (1), MASM does not use
13                                                      ;the current .ENABLE setting to
14                                                      ;resolve the referenced address.
15 000011     00000000005              A:      5
```

References

"Data" - Chapter 2
"Data Placement" - Chapter 6
"Expressions" - Chapter 2
"Relocation Bases" - Chapter 3
"Resolving Locations in MRI Instructions" - Chapter 3

End-of-program indicator .END

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 a .END (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
        .NREL
START:  SUB     0,0
                .
                .
                .       ;End of module MOD1.  Begin execution
        .END    START   ;of program at location START.
```

References

AOS/VS Link and Library File Editor (LFE) User's Manual - end blocks
"Expressions" - Chapter 2
"File Termination" - Chapter 6

End of conditional or repetitive assembly .ENDC

Syntax

.ENDC

.ENDC□**numeric-symbol**

Purpose

The .ENDC pseudo-op terminates a group of source lines whose assembly is repetitive (.DO) or conditional (.IFE, .IFG, .IFL, .IFN).

If the optional **numeric-symbol** is supplied, the .ENDC terminates assembly of lines following the .DO or .IFx and suppresses the assembly of all lines following the .ENDC up to the occurrence of **numeric-symbol** enclosed in square brackets.

In effect, the .ENDC with **numeric-symbol** provides a mechanism for if-then-else conditional assembly constructs. However, AOS/VS MASM provides an explicit .ELSE pseudo-op, whose use is recommended. .ENDC with **numeric-symbol** is provided primarily for compatibility with other Data General assemblers.

Example

```
        .DO     2
        1               ; These lines are assembled twice
        2
        .ENDC
        .
        .
        .IFN    1
        1               ; This is assembled
        .ENDC
        .
        .
        .IF     1
        2               ; This is assembled
        .ENDC   L1
        3               ; This is not assembled
[L1]    .
        .
```

Define an external entry .ENT

Syntax

.ENT□**numeric-symbol**

Purpose

The .ENT pseudo-op declares **numeric-symbol** as a symbol that you define in this source module but that you can refer to from separately assembled modules.

You must define a **numeric-symbol** in the module containing the .ENT declaration (see "Numeric Symbols" in Chapter 2). This numeric 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 refer to **numeric-symbol** from a separately assembled module, use one of the following pseudo-ops:

.EXTD .EXTN .EXTL

Example

```
        .TITLE  A
        .ENT    PTR     ;PTR is defined in this module and
                        ;can be referred to by other modules.
        .ZREL
PTR:    TABLE
        .NREL
TABLE:  0
                .
                .
                .
        .END


        .TITLE  B       ;Separately assembled module B.
        .EXTD   PTR     ;PTR is defined in another module.
        .NREL
        LWLDA   0,@PTR  ;Reference to externally defined symbol PTR.
                .
                .
                .
        .END
```

References

"Intermodule Communication" - Chapter 6
"Numeric Symbols" - Chapter 2

Define an overlay entry symbol .ENTO

Syntax

.ENTO□**numeric-symbol**

Purpose

The .ENTO pseudo-op is used when a 16-bit program is using overlays but is *not* using the AOS or AOS/VS 16-bit resource manager. When linked, **numeric-symbol** has a value which is a combination of the overlay area and overlay number of the module and partition it was defined in. Other modules can refer to **numeric-symbol** as an external (.EXTN) and use its value to load the overlay it was defined in.

Note that AOS and AOS/VS allow both shared and unshared overlays. So that **numeric-symbol** is associated with the correct overlay, the .ENTO statement must *follow* the corresponding data placement pseudo-op.

Example:

```
.TITLE   A
.NREL    1
.ENTO    OVERA
.END

.TITLE   B
.EXTN    OVERA
.NREL    1
LDA      0,=OVERA
ADC      1,1
?OVLOD            ; AOS system call
.END
```

Explicit end-of-file .EOF, .EOT

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.

NOTE: *.EOT is functionally equivalent to .EOF. It is provided for compatibility reasons.*

Example

```
        .TITLE  A       ;The first piece of source code
        .NREL           ;resides in file A.
START:          .
                .
                .
        .EOF            ;End of file A but not of source code.



        .NREL           ;The second part of the source
                .       ;code resides in file B.
                .
                .
        .END    START   ;End of current assembly.  Start program
                        ;execution at location START.
```

The MASM command line that assembles these two source modules is

```
) XEQ MASM A B ↵
```

Reference

"File Termination" - Chapter 6

Report a user-defined error message .ERROR

Syntax

.ERROR□**text**

Purpose

The .ERROR pseudo-op directs the macroassembler to treat the source line on which it appears as if it contained an error. Thus, a user-defined error message may appear in the error file.

Example

```
.
.
.DO      .ARGCT<4
.ERROR   Too few arguments passed to macro
.ENDC
.
.
```

Set macro escape character .ESC

Syntax

.ESC

.ESC□**abs-expr**

Purpose

The .ESC pseudo-op changes the macro escape character. The default macro escape character is an underscore (_), ASCII code 137_8. Within the macro definition string the escape character directs the assembler to store the next character without interpreting it. See the "Macro Definition" section of Chapter 5 for examples of the macro escape character. .ESC alone disables the macro escape character (giving it the value -1). .ESC **abs-expr** sets the macro escape character to the value of **abs-expr**. **Abs-expr** must be within the range of printable ASCII values.

You must take care when setting the macro escape character within a macro not to set it to current value using the single ASCII character to octal conversion format:

Wrong	**Correct**
`.MACRO SET_ESC`	`.MACRO SET_ESC`
`.ESC "-`	`.ESC`
`%`	`.ESC "-`
	`%`
`. . .`	`. . .`
`.ESC "-`	`.ESC "-`
`. . .`	`. . .`
`SET_ESC`	`SET_ESC`

The first example causes an error because the second tilde (-) is ignored and " by itself does not produce a valid 8-bit value. In the second example the macro escape character is disabled. The tilde (-) is not ignored this time as it is no longer the current macro escape character. "- produces a valid 8-bit value.

Define a chain-link external symbol .EXTC

Syntax

.EXTC□**numeric-symbol**

Purpose

Declares **numeric-symbol** to be a chain-link external. Each data location that refers to **numeric-symbol** will receive the address of the previous reference to **numeric-symbol**. In effect, chain-link externals may be used to build single-linked backwards lists of memory locations.

If **numeric-symbol** is declared as a chain-link external in more than one module, the backwards list will run through all modules. Note that no corresponding entry declaration should be made for **numeric-symbol**. All modules must declare it as a chain-link external, or Link will report an error.

Example

```
.TITLE   A
.EXTC    X
.
.
X                ; First reference to X
.
.
X                ; Value is the address of first reference to X
.
.
.END

.TITLE   B
.EXTC    X
.
.
X                ; Value is the address of second reference to X
.
.
.END
```

Define an external displacement (8–bit) reference

.EXTD, .EXTDA, .EXTDAN, .EXTDAW

Syntax:

.EXTD□**numeric-symbol**

.EXTDA□**numeric-symbol**

.EXTDAN□**numeric-symbol**

.EXTDAW□**numeric-symbol**

Purpose:

The .EXTDx pseudo-ops declare **numeric-symbol** as a symbol whose definition appears in a separately assembled module. You must declare **numeric-symbol** with a .ENT or similar pseudo-op in the module that defines it. You cannot redefine **numeric-symbol** as any other kind of symbol in the current assembly.

A symbol declared with the .EXTDx pseudo-ops is normally used in an 8-bit memory reference field of an instruction, but larger memory reference fields can also be used with .EXTD symbols.

The alternate forms of the .EXTD pseudo-op allow you to optionally specify the size of the data item addressed by **numeric-symbol**. The .EXTDAN specifies that the data item is 16 bits, while .EXTDAW specifies 32 bits.

Example

```
.EXTD    X
.
.
LDA      0,X
.

.
ELDA     0,X
.
.
```

Define an 8-bit data external .EXTDD

Syntax:

.EXTDD□**numeric-symbol**

Purpose:

The .EXTDD pseudo-op declares **numeric-symbol** as a symbol whose definition appears in a separately assembled module. You must declare **numeric-symbol** with a .ENT or similar pseudo-op in the module that defines it. You cannot redefine **numeric-symbol** as any other kind of symbol in the current assembly.

Symbols declared with the .EXTDD pseudo-op resolve to a constant, not an address. This distinction is important to MASM in how it relocates **numeric-symbol**. Thus, if you define an entry symbol in an assignment statement as -1, other modules, when referring to that symbol, should use the .EXTDD, .EXTND or .EXTLD pseudo-ops.

Example

```
.EXTDD   X
.
.
ANDI     X,0
.
.
```

Define a long (32-bit) external reference

.EXTL, .EXTLA, .EXTLAN, .EXTLAW

Syntax:

.EXTL□**numeric-symbol**

.EXTLA□**numeric-symbol**

.EXTLAN□**numeric-symbol**

.EXTLAW□**numeric-symbol**

Purpose:

The .EXTLx pseudo-ops declare **numeric-symbol** as a symbol whose definition appears in a separately assembled module. You must declare **numeric-symbol** with a .ENT or similar pseudo-op in the module that defines it. You cannot redefine **numeric-symbol** as any other kind of symbol in the current assembly.

A symbol declared with the .EXTLx pseudo-ops is normally used in a 31-bit memory reference field of an instruction.

The alternate forms of the .EXTL pseudo-op allow you to optionally specify the size of the data item addressed by **numeric-symbol**. The .EXTLAN specifies that the data item is 16 bits, while .EXTLAW specifies 32 bits.

Example

```
.EXTL    X
.EXTLAN  XN
.EXTLAW  XW

LNLDA    0,X
LWLDA    0,X
LNLDA    0,XN
LWLDA    1,XW
```

Define a 32-bit data external .EXTLD, .EXTG

Syntax:

.EXTLD□**numeric-symbol**

.EXTG□**numeric-symbol**

Purpose:

The .EXTLD and .EXTG pseudo-ops declare **numeric-symbol** as a symbol whose definition appears in a separately assembled module. You must declare **numeric-symbol** with a .ENT or similar pseudo-op in the module that defines it. You cannot redefine **numeric-symbol** as any other kind of symbol in the current assembly.

Symbols declared with the .EXTLD pseudo-op resolve to a constant, not an address. This distinction is important to MASM in how it relocates **numeric-symbol**. Thus, if an entry symbol is defined in an assignment statement as -1, other modules, when referring to that symbol, should use the .EXTDD, .EXTND or .EXTLD pseudo-ops.

Symbols declared with the .EXTG pseudo-op can be addresses or constants, and are treated as 32-bit integers, disregarding all address range checking.

Example

```
.EXTLD   X
.
.
WANDI    X,0
.
```

Define an external narrow (16-bit) reference

.EXTN, .EXTNA, .EXTNAN, .EXTNAW

Syntax:

.EXTN□**numeric-symbol**

.EXTNA□**numeric-symbol**

.EXTNAN□**numeric-symbol**

.EXTNAW□**numeric-symbol**

Purpose:

The .EXTNx pseudo-ops declare **numeric-symbol** as a symbol whose definition appears in a separately assembled module. You must declare **numeric-symbol** with a .ENT or similar pseudo-op in the module that defines it. You cannot redefine **numeric-symbol** as any other kind of symbol in the current assembly.

A symbol declared with the .EXTNx pseudo-ops is normally used in a 15-bit memory reference field of an instruction, but larger memory reference fields can also be used with .EXTN symbols.

The alternate forms of the .EXTN pseudo-op allow you to optionally specify the size of the data item addressed by **numeric-symbol**. The .EXTNAN specifies that the data item is 16 bits, while .EXTNAW specifies 32 bits.

Example

```
.EXTN    X
.EXTNAN  XN
.EXTNAW  XW

XNLDA    0,X
XWLDA    1,X
XNLDA    0,XN
XWLDA    1,XW
```

Define a 16-bit data external .EXTND

Syntax

.EXTND□**numeric-symbol**

Purpose

The .EXTND pseudo-op declares **numeric-symbol** as a symbol whose definition appears in a separately assembled module. You must declare **numeric-symbol** with a .ENT or similar pseudo-op in the module that defines it. You cannot redefine **numeric-symbol** as any other kind of symbol in the current assembly.

Symbols declared with the .EXTND pseudo-op resolve to a constant, not an address. This distinction is important to MASM in how it relocates **numeric-symbol**. Thus, if an entry symbol is defined in an assignment statement as -1, other modules, when referring to that symbol, should use the .EXTDD, .EXTND or .EXTLD pseudo-ops.

Example

```
.EXTND   X
.
.
ANDI     X,0
```

Treat undefined symbols as externals .EXTU

Syntax

.EXTU

Purpose

This pseudo-op causes the assembler to treat all symbols that are undefined after pass 1 as if they were externals. In effect, MASM assumes that all undefined symbols will be defined in other modules.

NOTE: *We don't suggest you use this pseudo-op. It is provided primarily for compatibility with other Data General assemblers.*

Force-link a module from a library .FORCE

Syntax

.FORCE

Purpose

The .FORCE 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 .FORCE in a module that resides in a library, that module will be linked whenever the library name occurs in a Link command line.

NOTE: *.FORC is an acceptable abbreviation of this pseudo-op.*

Example

```
        .TITLE  SQUARE  ;SQUARE is part of a library.  Whenever the
        .FORCE          ;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 that symbol is not
                .       ;referred to.
CUBE:           .
        .END
```

Reference

AOS/VS Link and Library File Editor (LFE) User's Manual – libraries

Create data word with word relocation .GADD

Syntax:

.GADD□**numeric-symbol**□**expression**

Purpose:

This pseudo-op generates a one-word data item whose contents are the 16-bit sum of the value of **numeric-symbol** and the **expression**. **Numeric-symbol** is assumed to be an external, and **expression** is assumed to be absolute.

This pseudo-op is primarily for the use of 16-bit programs, and is provided for compatibility with other Data General assemblers. AOS/VS MASM permits you to write

.WORD **numeric-symbol + expression**

which is equivalent to .GADD.

Example

```
        .EXTN   X
        .
        .
L1:     .GADD   X,1B0
        .
        .
        LDA     1,L1
```

Generate gate entry with ring field .GATE

Syntax

.GATE␣**numeric-symbol**␣**expression**

Purpose

The .GATE pseudo-op permits you to initialize entries for MV/Family hardware gate arrays. **Numeric-symbol** can be any label or external address, and **expression** is an absolute expression giving the highest ring number that the gate entry is valid for. (The ring number must be shifted into bits 1-3.) Link will then remove the ring from **numeric-symbol**'s address and replace it with the ring in **expression**.

Example

```
          .EXTL    GATEE

          .GATE    GATEE,7S3
          .GATE    LABEL,5S3

          .NREL    1
LABEL:    .
          .
          .
```

Set the current location counter .GLOC

Syntax

.GLOC□**expression**

Purpose

The .GLOC pseudo-op sets the location counter to the value and relocation base given by the **expression**. If **expression** is absolute, data following the .GLOC is absolute. If **expression** is relocatable, data following the .GLOC is relocatable relative to the base of the **expression**.

In AOS/VS MASM, this pseudo-op is the functional equivalent of the .LOC pseudo-op directive. It is provided primarily for compatibility with other Data General assemblers.

Example

```
.EXTN    X
.
.
.GLOC    X+5
1
2
.
.
```

Jump ahead in conditional assembly .GOTO

Syntax

.GOTO□**numeric-symbol**

Purpose

This pseudo-op suppresses the assembly of lines until MASM encounters an occurrence of **numeric-symbol** inside brackets.

NOTE: *We do not suggest you use this pseudo-op.*

Example

```
          .
          .
          LDA     0,1,2    ; Assembled instruction
          STA     2,1,0    ; Assembled instruction

          .GOTO   LABEL

          LDA     1,1,1    ; Unassembled instruction
          STA     2,2,2    ; Unassembled instruction

[LABEL]   LDA     1,1,1    ; Assembly resumes
          .
          .
```

Create data word with GREF relocation .GREF

Syntax

.GREF□**numeric-symbol**□**expression**

Purpose

The .GREF pseudo-op is similar to the .GADD pseudo-op, the only difference being that the high-order bit (bit 0) of the **expression** is preserved. In other words, .GREF performs a 15-bit addition between **numeric-symbol** and **expression**, while .GADD performs a 16-bit addition.

This pseudo-op is for use primarily in 16-bit programs, and is provided for compatibility with other Data General assemblers.

Example

```
        .EXTN   X
        .
        .
L1:     .GREF   X,1B0+5  ; Set indirect bit in data word
        .                ; which will cause a second level
        .                ; of indirection
        LDA     0,@L1
        .
        .
```

Perform conditional assembly .IFE, .IFG, .IFL, .IFN

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 *not* satisfy that condition, MASM assembles the source lines following the .ELSE pseudo-op, if one is present.

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□**abs-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.
```

.IFE, .IFG, .IFL, .IFN (continued)

If you include the optional .ELSE pseudo-op, your source code will appear as follows:

```
.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.
.
.ELSE
.                ;Assemble these source lines if abs-expr
.                ;does not satisfy the .IFx condition.
.
.ENDC            ;Terminate conditional portion of source.
```

You can 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 (if we ignore the possibility of an .ELSE block). For example, the statement .IFE A is equivalent to .DO A==0. Both 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.

Examples

```
01                00000000000               A=0
02                00000000001               .IFE    A       ;A equals 0 so MASM assembles the
03 000000         00000000100               100             ;conditional portion of the module.
04 000002         00000000200               200
05                                          .ENDC           ;End of conditional.
```

```
01                00000000000               A=0
02                00000000000               .IFG    A       ;A is not greater than 0 so MASM
03                                          100             ;assembles the .ELSE portion of the
04                                          200             ;conditional
05                                          .ELSE
06 000000         00000000300               300
07 000002         00000000400               400
08                                          .ENDC           ;End of conditional.
```

References

"Absolute Expressions" - Chapter 3
"Loops and Conditionals in Macros" - Chapter 5
"Repetitive and Conditional Assembly" - Chapter 6

Create a data word with call relocation .KCALL

Syntax

.KCALL[□**numeric-symbol**]

Purpose

The .KCALL pseudo-op generates one data word with a value of 6013_8 and call relocates it relative to **numeric-symbol**, if present. **Numeric-symbol** must be an external (.EXTN) symbol, and should resolve to a .PENT symbol.

.CALL, along with .TARG, generates call and target data words for using the AOS and AOS/VS 16-bit resource manager. The Link utility resolves the contents of call and target words based on the call type and the program's overlay structure. See the *AOS/VS Link and Library File Editor (LFE) User's Manual* for a detailed discussion of resource calls.

Example

```
.EXTN    P
.
.
.KCALL   P        ; ?KCALL resource call word
.TARG    0,P      ; ?KCALL resource target word
.
.
```

Begin listing page with Data General proprietary header .LCNS

Syntax

.LCNS

Purpose

The .LCNS pseudo-op directs the assembler to add the Data General proprietary software notice to the top of each page of the listing. This pseudo-op is for use by Data General personnel.

Set the current location counter .LOC

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 can 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 can use .LOC as a value symbol, in which case it has the value and relocation property of the current location counter.

For example, using .PUSH and .POP, you can save and restore the value and relocation base of the location counter as follows:

```
.PUSH    .LOC     ;Save the value and relocation base
.                 ;of the location counter on the stack.
.
.
.LOC     .POP     ;Set the value and relocation base of
                  ;the location counter equal to the entry
                  ;on the top of the stack.
```

Note that the .LOC and period (.) value symbols are identical.

.LOC (continued)

Example

```
01                00000000000                    .NREL   0       ;Unshared code partition (NREL)
02 000000 UC      00000000001          N:        1
03 000002 UC      00000000002                    2
04 000004 UC      00000000003                    3
05                00000000100                    .LOC    100     ;Set the location counter to
06 000100         00000000004                    4               ;absolute location 100.
07 000102         00000000005                    5
08 000104         00000000006                    6
09                00000000050  UC                .LOC    N+50    ;Set the location counter to
10 000050 UC      00000000007                    7               ;the relocation base of N
11 000052 UC      00000000010                    10              ;(unshared NREL code) and
12 000054 UC      00000000011                    11              ;start assigning locations
13                                                               ;at the 50th address after N.
14
```

References

"Assigning Locations" - Chapter 3
"Expressions" - Chapter 2
"Location Counter" - Chapter 3

Create a literal pool .LPOOL

Syntax

.LPOOL

Purpose

This pseudo-op directs the assembler to create a literal pool of all the currently outstanding literals. The pool is created in the partition and the location where the .LPOOL pseudo-op occurs. Make sure that the literal pool is not in the path of execution.

NOTE: *.LPOO is an acceptable abbreviation of this pseudo-op.*

Example

```
.NREL
.
.
LDA     1,=2     ; 2 goes into literal pool
LDA     1,=.     ; Value of . goes into literal pool
.
.
RTN
.LPOOL           ; Value of 2 and . go here
.
.
```

Define a macro .MACRO

Syntax

.MACRO□**numeric-symbol**

macro-definition-string

%

Purpose

The .MACRO pseudo-op defines **numeric-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 **numeric-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 (**%**). **%** must appear in the first column of its own source line.

Chapter 5 provides a complete discussion of the MASM macro facility.

NOTE: *.MACR is an acceptable abbreviation of this pseudo-op.*

Example

```
01                00000000000                 .NREL   0
02                                            .MACRO  TEST    ;Define macro TEST.
03                                            ^1              ;Macro definition consists of 3
04                                            ^2              ;data statements that get their
05                                            ^3              ;values from the first 3 arguments
06                                    %                       ;passed to TEST
07
08
09                                            TEST    4,5,6   ;Call TEST with 3 arguments.
10 000000 UC  00000000004                     4               ;Macro definition consists of 3
11 000002 UC  00000000005                     5               ;data statements that get their
12 000004 UC  00000000006                     6               ;values from the first 3 arguments
13
14                                            TEST    0,1,2   ;Call TEST with 3 more arguments.
15 000006 UC  00000000000                     0               ;Macro definition consists of 3
16 000010 UC  00000000001                     1               ;data statements that get their
17 000012 UC  00000000002                     2               ;values from the first 3 arguments
18
19
```

References

"Macros" - Chapter 5
"Numeric Symbols" - Chapter 2

Indicate macro usage .MCALL

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

```
01                                          .ZREL
02 000000 ZR  00000000100                   TABLE1: .BLK    100
03 000100 ZR  00000000100                   TABLE2: .BLK    100
04 000200 ZR  00000000000  ZR               LOC1:   TABLE1
05 000202 ZR  00000000100  ZR               LOC2:   TABLE2
06
07            00000000000                   .NREL   0
08
09                                          .MACRO  MC      ;Define macro MC.
10                                          .IFE    .MCALL  ;On the first call to macro MC,
11                                          LDA     0,LOC1  ;assemble the first LDA instr.
12                                          .ELSE           ;On subsequent calls to macro MC,
13                                          LDA     0,LOC2  ;assemble the second LDA instr.
14                                          .ENDC           ;End of conditional.
15                                  %                       ;End of macro definition.
16
17
18                                          MC              ;First call to MC (.MCALL equals 0).
19            00000000001                   .IFE    .MCALL  ;On the first call to macro MC,
20 000000 UC 020200         ZR              LDA     0,LOC1  ;assemble the first LDA instr.
21                                          .ELSE           ;On subsequent calls to macro MC,
22                                          LDA     0,LOC2  ;assemble the second LDA instr.
23                                          .ENDC           ;End of conditional.
24
25                                          MC              ;Second call to MC (.MCALL equals 1).
26            00000000000                   .IFE    .MCALL  ;On the first call to macro MC,
27                                          LDA     0,LOC1  ;assemble the first LDA instr.
28                                          .ELSE           ;On subsequent calls to macro MC,
29 000001 UC 020202         ZR              LDA     0,LOC2  ;assemble the second LDA instr.
30                                          .ENDC           ;End of conditional.
31
```

Reference

"Macro-Related Pseudo-Ops" - Chapter 5
"Macros" - Chapter 5

Place literal pool in NREL .NLIT

Syntax

.NLIT

Purpose

The .NLIT pseudo-op directs the assembler to place the default literal pool in whatever partition is active at the end of the assembly pass. Normally, the default literal pool is placed in ZREL. This pseudo-op cannot be used subsequent to an actual literal definition, and is usually placed at the beginning of the source file.

Example

```
.TITLE   A
.NLIT
.NREL    1
LDA      0,=2        ; 2 goes into literal pool
.END                 ; Literal pool is placed in NREL 1
```

Inhibit or enable the listing of conditional lines — .NOCON

Syntax

.NOCON□**abs-expr**

Purpose

The .NOCON pseudo-op either inhibits or permits the listing of the conditional source module code that does 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 can override the .NOCON pseudo-op at assembly time by using the /XPAND function switch on the MASM command line. Refer to Chapter 8 for more information.

NOTE: *.NOCO is an acceptable abbreviation of this pseudo-op.*

Value

You can 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

Consider the following source code:

```
A=3
.IFE    A       ;False.  MASM lists false
10              ;conditionals, by default.
20
30
.ENDC

.NOCON  1       ;Inhibit listing of false conditionals.

.IFE    A       ;False.  MASM will not list this portion
40              ;of code.
50
60
.ENDC
```

.NOCON (continued)

```
        .IFN    A       ;True.  MASM lists the assembled code
        70              ;regardless of the .NOCON setting.
        100
        110
        .ENDC
```

The assembly listing for this portion of code is:

```
01              00000000003             A=3
02              00000000000             .IFE    A       ;False.  MASM lists false
03                                      10              ;conditionals, by default.
04                                      20
05                                      30
06                                      .ENDC
07
08              00000000001             .NOCON  1       ;Inhibit listing of false conditionals.
09
10              00000000000             .IFE    A       ;False.  MASM won't list this portion.
11
12              00000000001             .IFN    A       ;True.  MASM lists the assembled code
13 000000       00000000070             70              ;regardless of the .NOCON setting.
14 000002       00000000100             100
15 000004       00000000110             110
16                                      .ENDC
17
18
```

References

"Absolute Expressions" - Chapter 3
"Assembly Listing" - Chapter 4
"Command Line Switches" (/XPAND) - Chapter 8
"Repetitive and Conditional Assembly" - Chapter 6

Inhibit or enable the listing source lines without location fields — .NOLOC

Syntax

.NOLOC□**abs-expr**

Purpose

The .NOLOC pseudo-op directs the assembler to either inhibit or enable the listing of source lines lacking 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 can override the .NOLOC pseudo-op at assembly time by using the /XPAND function switch on the MASM command line. Refer to Chapter 8 for more information.

NOTE: *.NOLO is an acceptable abbreviation of this pseudo-op.*

Value

You can 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 lines,
.NREL   1       ;by default.
.TXT   "ABCDEF"


.NOLOC  1       ;Inhibit listing of assembly lines
                ;that lack location fields.


.TXT   "GHIJKL" ;MASM only lists the first line.
LWLDA   0,x     ;Listed.
.LOC    .+5     ;Not listed.
x:      5       ;Listed.
.END            ;Not listed.
```

.NOLOC (continued)

The assembly listing for this source is:

```
01                                      .TITLE  DF      ;MASM lists all source lines,
02             00000000001              .NREL   1       ;by default.
03 000000 SC 040502 041504              .TXT  "ABCDEF"
04           042506 000000
05
06 000004 SC 043510 044512              .TXT  "GHIJKL"  ;MASM only lists the first line.
07 000010 SC 121771                     LWLDA   0,X     ;Listed.
08 000020 SC  00000000005               X:      5       ;Listed.
```

References

"Absolute Expressions" - Chapter 3
"Assembly Listing" - Chapter 4
"Command Line Switches" (/XPAND) - Chapter 8

Inhibit or enable the macro expansions — .NOMAC

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 can override the .NOMAC pseudo-op at assembly time by using the /XPAND function switch on the MASM command line. Refer to Chapter 8 for more information.

NOTE: *.NOMA is an acceptable abbreviation of this pseudo-op.*

Value

You can 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

```
01                                      .MACRO  MAC     ;Define macro MAC.
02                                      5
03                                      6
04                                  %
05
06
07                                      MAC             ;Call macro MAC.  By default,
08 000000     00000000005               5
09 000002     00000000006               6
10                                                      ;MASM lists the expansion.
11
12            00000000001               .NOMAC  1       ;Inhibit listing of macro expansions.
13
14                                      MAC             ;MASM does not list MAC's expansion.
15
16 000010     00000000100               100             ;Assemble the value 100.
```

.NOMAC (continued)

Our second example shows how to use .NOMAC inside a macro definition string.

```
01                                      .MACRO  INSIDE  ;Define macro INSIDE.
02                                      2
03                                      3
04                                      .NOMAC  1       ;During expansion of macro
05                                      4               ;INSIDE, MASM does not list
06                                      5               ;data statements 4 and 5.
07                                      .NOMAC  0       ;Re-enable listing of macro
08                                      6               ;expansions.
09                                      7
10                              %
11
12                                      INSIDE          ;Call to macro INSIDE.
13 000000       00000000002             2
14 000002       00000000003             3
15              00000000001             .NOMAC  1       ;During expansion of macro
16 000010       00000000006             6               ;expansions.
17 000012       00000000007             7
18
```

References

"Absolute Expressions" - Chapter 3
"Assembly Listing" - Chapter 4
"Command Line Switches" (/XPAND) - Chapter 8
"Listing of Macro Expansions" - Chapter 5

Set location counter to a default NREL partition .NREL

Syntax

.NREL[□**expression**]

Purpose

The .NREL pseudo-op directs that subsequent data and/or code be placed in one of the default NREL partitions, up to the next .GLOC, .LOC, .NREL, .PART or .ZREL pseudo-op.

Using .NREL, you can specify any of five predefined memory partitions. These partitions provide for the four combinations of the data/code and shared/unshared attributes, plus a fifth partition with the short attribute. Table 7-2 lists the NREL memory partitions and their attributes

Table 7-2 NREL Partitions

Partition	Attributes
Low Data	Short, unshared, data, 1-word alignment normal base, overwrite-with-message
Unshared Data	Long, unshared, data, 1-word alignment normal base, overwrite-with-message
Unshared Code	Long, unshared, code, 1-word alignment normal base, overwrite-with-message
Shared Data	Long, shared, data, 1-word alignment normal base, overwrite-with-message
Shared Code	Long, shared, code, 1-word alignment normal base, overwrite-with-message

Short partitions are confined to the first 77777_8 words of the program's address space. Long partitions can be positioned anywhere. The shared partitions are normally write protected during execution, so more than one process can share a single copy of the shared partitions. The data partitions usually contain nonexecutable portions of the program. Partitions with the code attribute usually contain both executable and nonexecutable parts of the program.

You specify which of these prefined partitions to use with the value of the expression. The expression must be absolute, and it must have one of the values shown in Table 7-3.

.NREL (continued)

Table 7-3 NREL Partition Numbers

Partition	Number
Low Data	.NREL 2
Unshared Data	.NREL 6
Unshared Code	.NREL 4, .NREL 0, or .NREL
Shared Data	.NREL 5
Shared Code	.NREL 7 or .NREL 1

Example

```
        .
        .
        .NREL   2       ; Low data
L1:     0

        .NREL   6       ; Unshared data
L2:     0

        .NREL   7       ; Shared code
        LWADI   1,L2    ; L1 is unshared, not write protected
                        ; and so may be modified.
        XWADI   1,L1    ; A 16-bit displacement can be used to
                        ; address data in NREL 2.
        .
        .
```

Name an object file .OB

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 and 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 /O= switch on the MASM command line, the assembler overrides the .OB pseudo-op, and the object file receives the name following the /O= switch.

NOTE: *Older assembly language sources may use .RB in place of .OB. The two pseudo-ops are interchangeable.*

In summary, the assembler uses the following hierarchy to name object files:

Priority	Object Filename	Description
1 (highest)	/O=filename	The switch on the MASM command line
2	.OB□filename	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 can direct the assembler to assign a name to the object file according to the evaluation of some expression (see the example).

.OB (continued)

Example

```
.IFE     VAR      ;If the value of VAR equals 0, MASM
.OB      SYS1     ;names the object file SYS1.OB.
.ELSE             ;Otherwise, MASM names the
.OB      SYS2     ;file SYS2.OB.
.ENDC
```

References

"Command Line Switches" (/N and /O=) - Chapter 8
"Object File" - Chapter 4

Create a named relocatable memory partition .PART

Syntax

.PART□**numeric-symbol**[□**attribute**]

Purpose

Use the .PART pseudo-op to create a relocatable memory partition, i.e., a contiguous portion of memory with a name and attributes. All code and data that follow .PART, up to the next .LOC, .GLOC, .NREL, .PART or .ZREL pseudo-op, will reside in that partition.

The assembler also makes available a number of predefined relocatable partitions through the .NREL and .ZREL pseudo-ops. These are unnamed, and are always the same. The .PART pseudo-op permits the creation of partitions with name **numeric-symbol**, and with a user-defined combination of attributes. The attributes are:

NREL or ZREL
LONG or SHORT
UNSHARED or SHARED
CODE or DATA
NORM or COMM
GLOBAL or LOCAL
MESS or NOMESS

In addition, you can specify the alignment of the partition as an attribute.

Table 7-4 describes the various attribute arguments. Specify the desired attributes after the partition name in the .PART statement. If an attribute and its converse are both omitted, MASM uses the default for that attribute. The defaults are:

NREL, LONG, UNSHARED, CODE, NORM, GLOBAL, MESS, ALIGN=0

.PART (continued)

Table 7-4 .PART Attribute Arguments

Attribute	Description
NREL or ZREL	An NREL partition is normally allocated memory space by Link at address 400_8 and above. A ZREL partition is normally allocated below address 400_8.
LONG or SHORT	A long partition can reside anywhere in memory. A short partition is confined to memory addresses below 100000_8.
UNSHARED or SHARED	An unshared partition can be read, executed, or written to and modified. A shared partition is write-protected; it can not be modified. AOS/VS places all shared partitions at higher addresses than unshared partitions.
CODE or DATA	A code partition can contain either code or data. A data partition normally contains only data.
NORM or COMM	If two modules have code or data in the same normal partition, that code or data will be allocated sequential addresses in memory. If two modules have code or data in the same common partition, that code or data will overlay each other at the same addresses.
GLOBAL or LOCAL	A global partition's name is visible to all other separately compiled modules. Thus all global partitions with the same name are the same partition. A local partition's name is not visible to other modules. Therefore, that partition can be referred to by only that module.
MESS or NOMESS	Normally, Link reports an error message when two modules' data overwrite each other. This Link warning message can be suppressed with NOMESS.
ALIGN=n	Any partition can be aligned to a particular memory boundary, e.g., double-word alignment, block alignment or page alignment. This alignment, n, is given as a power of two, and the partition's address will be a multiple of 2^n.

If you leave a .PART partition at one point in an assembly and you want to place more data in that partition at a later point, use a new .PART statement with the same partition name.

If you want to place code or data from separately assembled modules into the same partition (see example, next page), use another .PART statement with the same partition name. (Obviously, these separately assembled partitions must not have the LOCAL attribute.) When you link these modules together, the code or data from both is placed in partition TJ. If the partitions have the COMM attribute, the code or data from each module is placed on top of each other. Otherwise, it is placed consecutively in the partition.

If a partition is given different attributes in two .PART statements, MASM and LINK attempt to resolve the attributes and report an error if they cannot be resolved. The attributes are resolved as follows:

.PART (continued)

NREL and ZREL	Resolved in favor of ZREL
LONG and SHORT	Resolved in favor of SHORT
UNSHARED and SHARED	An error is reported
CODE and DATA	An error is reported
NORM and COMM	An error is reported
GLOBAL and LOCAL	An error is reported
MESS and NOMESS	Resolved in favor of NOMESS
ALIGN differences	Resolved in favor of the higher alignment

Example

```
.PART   TJ      ALIGN=1,SHORT
.
.                                   ; This is code in partition TJ
.ZREL
.
.                                   ; This is code in ZREL
.PART   TJ      ALIGN=1,SHORT
.
.                                   ; This is also code in partition TJ
```

Number of assembly pass .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. Since the values of A, B, and C remain constant, MASM need not assemble them on pass two. (This use of .PASS is similar to the /PASS1 local switch described in Chapter 8.)

```
.IFE     .PASS
A=0              ;MASM assembles these statements
B=1              ;on pass one only (i.e., when
C=2              ;.PASS equals 0).
.ENDC
```

Define a procedure entry symbol .PENT

Syntax

.PENT□**numeric-symbol**

Purpose

This pseudo-op is similar to the .ENT pseudo-op, but defines **numeric-symbol** as type PENT, rather than type ENTRY. The difference is important to 16-bit programs using the AOS or AOS/VS 16-bit resource manager. PENT symbols form the relocation bases for CALL and TARGET relocations as generated by the .CALL, .KCALL, .RCALL, .RCHAIN, .TARG and .PTARG pseudo-ops.

Example

```
        TITLE   A
        .NREL   1
        .
        .
        ?RSAVE  0
        .
        .
        ?RCALL  B
        .
        .
        .END

        TITLE   B
        .PENT   B
        .NREL   1
        .
        .
B:      ?RSAVE  0
        .
        .
        RTN
```

Pop the value and relocation property of last expression pushed onto the stack .POP

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 the stack.

If the assembler stack contains no values, then .POP has the value 0 and the absolute relocation property. In addition, MASM returns an error for that source statement.

Example

```
01              00000000025             A=25            ;Define A.
02 000000       00000000025             A               ;Assemble A's present value.
03
04                                      .PUSH   A       ;Push A's value onto the assembler's
05              00000000015             A=15            ;stack.  Assign A a new value
06 000002       00000000015             A               ;and assemble that value.
07
08              00000000025             A=.POP          ;Assign A the value on the top
09 000004       00000000025             A               ;of the stack and assemble A's
10                                                      ;new value.
```

See the .PUSH description for another example.

References

"Relocatability" - Chapter 3
"Stack Control" - Chapter 6

Create a data word with target relocation

.PTARG

Syntax

.PTARG□**numeric-symbol**

Purpose

The .PTARG pseudo-op generates one data word with a value of 2, and target relocates it relative to **numeric-symbol**. **Numeric-symbol** must be an external (.EXTN) symbol, and should resolve to a .PENT symbol.

The .PTARG pseudo-op generates target data words for using the AOS and AOS/VS 16-bit resource manager. The Link utility resolves the contents of target words based on the value of the data word and the program's overlay structure. See the *AOS/VS Link and Library File Editor (LFE) User's Manual* for a detailed discussion of resource calls.

NOTE: *.PTAR is an acceptable abbreviation of this pseudo-op.*

Example

```
        .EXTN   P
        .
        .
L1:     .PTARG  P
        .
        .
        LDA     0,L1    ; Load resource target
        PSH     0,0     ; Push onto stack

        ?RCALL          ; Make parametric resource call
        .
        .
```

Push a value and its relocation property onto the stack .PUSH

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 can 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

```
01                00000000010             .RDX    8       ;Input radix is 8 (i.e.,
02 000000         00000000010             (.RDX)          ;octal).  Assemble the current
03                                                        ;input radix value.
04                                        .PUSH   .RDX    ;Save the input radix on the
05                00000000012             .RDX    10      ;stack.  Set the input radix
06 000002         00000000012             (.RDX)          ;to 10 (i.e., decimal).
07                                                        ;Assemble the current input
08                                                        ;radix value.
09                00000000010             .RDX    .POP    ;Set the input radix to the
10                                                        ;value on top of the stack
11 000004         00000000010             (.RDX)          ;(in this case, 8).  Assemble
12                                                        ;the current input radix value.
```

See the .POP description for another example.

References

"Expressions" - Chapter 2
"Relocatability" - Chapter 3
"Stack Control" - Chapter 6

Create a data word with call relocation .RCALL

Syntax

.RCALL

.RCALL□**numeric-symbol**

Purpose

The .RCALL pseudo-op generates one data word with a value of 6014_8, and call relocates it relative to **numeric-symbol**, if present. **Numeric-symbol** must be an external (.EXTN) symbol, and should resolve to a .PENT symbol.

.RCALL, along with .TARG, generate call and target data words for using the AOS and AOS/VS 16-bit resource manager. The Link utility resolves the contents of call and target words based on the data word value and the program's overlay structure. See the *AOS/VS Link and Library File Editor (LFE) User's Manual* for a detailed discussion of resource calls.

NOTE: *.RCAL is an acceptable abbreviation of this pseudo-op.*

Example

```
.EXTN    P
.
.
.RCALL   P        ; ?RCALL resource call word
.TARG    1,P      ; ?RCALL resource target word
.
.
```

Create a data word with call relocation .RCHAIN

Syntax

.RCHAIN

.RCHAIN□**numeric-symbol**

Purpose

The .RCHAIN pseudo-op generates one data word with a value of 6015_8, and call relocates it relative to **numeric-symbol**, if present. **Numeric-symbol** must be an external (.EXTN) symbol, and should resolve to a .PENT symbol.

.RCHAIN, along with .TARG, generate call and target data words for using the AOS and AOS/VS 16-bit resource manager. The Link utility resolves the contents of call and target words based on the data word value and the program's overlay structure. See the *AOS/VS Link and Library File Editor (LFE) User's Manual* for a detailed discussion of resource calls.

NOTE: *.RCHA is an acceptable abbreviation of this pseudo-op.*

Example

```
.EXTN    P
.
.
.RCHAIN  P        ; ?RCHAIN resource call word
.TARG    2,P      ; ?RCHAIN resource target word
.
.
```

Set radix for numeric input conversion .RDX

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 is 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 7-5 shows the correspondence between numeric representations in various bases.

Table 7-5 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

.RDX (continued)

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).

Value

You can 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 can alter this setting with the .RDXO pseudo-op.

```
01              00000000010             .RDX    8       ;Input radix is 8.
02 000000       00000000123             123             ;MASM assembles 123 in octal.
03
04              00000000012             .RDX    10      ;Set input radix to 10.
05 000002       00000000173             123             ;MASM assembles 123 in decimal.
06
07              00000000020             .RDX    16      ;Set input radix to 16.
08 000004       00000000443             123             ;MASM assembles 123 in hexadecimal.
09 000006       00000000017             0F              ;Note the leading zero.
10 000010       00000000173             123.            ;MASM assembles 123 in decimal even
11                                                      ;though the input radix is 16.
12
13 000012       00000000020             (.RDX)          ;Assemble the current input radix
14                                                      ;value.
```

References

"Absolute Expressions" - Chapter 3
"Numbers" - Chapter 2
"Radix Control" - Chapter 6

Set the radix for numeric output conversion .RDXO

Syntax

.RDXO[□**abs-expr**]

Purpose

The .RDXO psuedo-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** in decimal. This argument must be an absolute expression and its value must be in the following range:

$$8_{10} \text{ <= abs-expr <= } 20_{10}$$

If you do not specify an output radix in your module, the assembler uses octal.

Table 7-5 in the .RDX pseudo-op description shows the correspondence between numeric representations in various bases.

Value

You can 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

```
01                00000000010          .RDX    8       ;Input radix is 8.
02                00000000010          .RDXO   8       ;Output radix is 8.
03 000000         00000000077          77
04 000002         00000000022          22
05 000004         00000000045          45
06                00000000012          .RDX    10      ;Input radix is 10.
07                00000000012          .RDXO   10      ;Output radix is 10.
08 000006         0000000077           77
09 000008         0000000022           22
10 000010         0000000045           45
11                0000000016           .RDX    16      ;Input radix is 16.
12                0000000016           .RDXO   16      ;Output radix is 16.
13 00000C         00000077             77
14 00000E         00000022             22
15 000010         00000045             45
16                00000008             .RDXO   8       ;Input radix is 16.
17 000022         00000000167          77              ;Output radix is 8.
18 000024         00000000042          22
19 000026         00000000105          45
20                00000000012          .RDXO   10      ;Input radix is 16.
21 000024         0000000119           77              ;Output radix is 10.
22 000026         0000000034           22
23 000028         0000000069           45
24 000030         0000000010           (.RDXO)         ;Assemble the value of the current
25                                                     ;output radix.  MASM always
26                                                     ;represents this as 10, regardless
27                                                     ;of the current base.
```

.RDXO (continued)

References

"Absolute Expressions" - Chapter 3
"Assembly Listing" - Chapter 4
"Numbers" - Chapter 2
"Radix Control" - Chapter 7

Set revision level .REV

Syntax

.REV□w[□x[□y[□z]]]

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 w, x, y, and z 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 a value for the w argument. The other three revision-level arguments are optional (Note, however, that you must supply x if you supply y, and you must supply both x and y if you supply z). MASM assigns the value 0 to all missing arguments.

For example, the following two .REV statements are equivalent:

```
.REV    1,0,0,0
.REV    1
```

Similarly, the following three source lines resolve to the same revision level:

```
.REV    3,2,0,0
.REV    3,2,0
.REV    3,2
```

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 REV command to obtain the revision level of any executable program file.

Example

```
.TITLE  MNTS
.REV    12,5,1,1 ;MASM interprets the revision level
.NREL            ;in the current input radix (octal,
                 ;by default).
```

References

"Absolute Expressions" - Chapter 3
AOS/VS Command Line Interpreter (CLI) User's Manual- REV command
AOS/VS Link and Library File Editor (LFE) User's Manual - /REV switch

Flag if previous instruction may skip .SKIP

Value

The .SKIP pseudo-op allows you to examine the state of the Skip flag which indicates whether the previous instruction may try to skip the next sequential word. This flag is 1 if the previous instruction was in one of the following classes:

.DIMS .DISS .DIWS .DLMS .DUNS .DUWS .DWMS .DXMS

In addition, this flag is set if a .DALC instruction has a skip option coded. In all other cases, .SKIP's value is zero.

The .SKIP pseudo-op can set the value of the Skip flag. The syntax for setting the flag is

.SKIP <n>

Example

```
.
.
XNISZ   T,3
.DO     .SKIP
NOP
.ENDC
.
.
```

Store an expression in one word and return an error if the high-order bits are not all ones or zeros

.SWORD

Syntax

.SWORD□**expr**

Purpose

Use the .SWORD pseudo-op to generate a 16-bit storage word with the value and relocation property of **expr**.

The assembler first computes the value of the expression in 32-bit mode. Then, the assembler truncates the leftmost 16 bits of the result and stores the remaining value in a single 16-bit word.

If, before truncation, the high-order 17 bits of the expression are *not* all ones or all zeros, the assembler returns an error. That is, the bits in the first word of the supplied expression must all equal the sign bit of the second word or you receive an error. MASM generates the 16-bit storage word regardless of whether it returns an error.

The above discussion explains the Macroassembler's actions if you supply .SWORD with an absolute expression. However, if you specify a relocatable expression, MASM cannot completely resolve it and, therefore, cannot perform the .SWORD operation. In this case, the truncation and error testing occur at link time, not assembly time.

Note that this pseudo-op does not alter the global data placement mode (set with .ENABLE).

.SWORD (continued)

Example

```
01 000000     00000000005              5               ;By default, each data value
02 000002     00000000006              6               ;occupies two words of memory.
03 000004    000005                    .SWORD  5,6     ;Store the values 5 and 6
04           000006
05                                                     ;in single words of memory.
06 000006    177255                    .SWORD  37777777255
07                                                     ;Truncate the first 16 bits
08                                                     ;(all ones) and store the
09                                                     ;result (177255) in a single
10                                                     ;word of memory.
 E 000007    065125                    .SWORD  1065125 ;This statement generates an
12                                                     ;error because the high-order
13                                                     ;bits are not all ones or zeros.
14 000010     00000000007              7               ;The global data placement mode
15 000012     00000000010              10              ;still specifies two words per
16                                                     ;data entry.
17                                     .END

pass 2 errors:
        EX7_42    1/10: listing   1/11: data value is out of range

1 ASSEMBLY ERROR
```

References

"Absolute Expressions" - Chapter 3
"Data Placement" - Chapter 6
"Expressions" - Chapter 2
"Relocatable Expressions" - Chapter 3

Create a data word with target relocation .TARG

Syntax

.TARG□**abs-expr**□**numeric-symbol**

Purpose

The .TARG pseudo-op generates one data word with a value of **abs-expr**, and target relocates it relative to **numeric-symbol**. **Numeric-symbol** must be an external (.EXTN) symbol, and should resolve to a .PENT symbol.

.TARG, along with .CALL, generates call and target data words for using the AOS and AOS/VS 16-bit resource manager. The Link utility resolves the contents of call and target words based on the value of **abs-expr** and the program's overlay structure. See the *AOS/VS Link and Library File Editor (LFE) User's Manual* for a detailed discussion of resource calls.

Example

```
.EXTN    P
.
.
.CALL    4,P      ; ?RCALL resource call word
.TARG    4,P      ; ?RCALL resource target word
.
.
```

Entitle an object module .TITLE

Syntax

.TITLE␣**numeric-symbol**

Purpose

The .TITLE pseudo-op provides a name for your object module by placing **numeric-symbol** in the module's title block (described in the *AOS/VS Link and Library File Editor (LFE) User's Manual*).

The title you assign in the module appears at the top of each page in the assembly listing. **Numeric-symbol** need not be unique (i.e., different from other symbols in your source module).

If you do not include .TITLE in your source module, the assembler supplies the default name ".MAIN."

Note that the name you assign in the .TITLE statement has no relation to the name of the object file (see the .OB pseudo-op).

NOTE: *.TITL is an acceptable abbreviation of this pseudo-op.*

Example

```
.TITLE  MODULE1
.
.
.
```

References

"Assembly Listing" - Chapter 4
AOS/VS Link and Library File Editor (LFE) User's Manual - title block
"Numeric Symbols" - Chapter 2

Return item from top of stack .TOP

Value

The permanent symbol .TOP has the value and relocation property of the last expression placed on the assembly-time stack (using .PUSH). .TOP differs from .POP in that .TOP does not pop the value from the stack.

If the assembler stack contains no values, MASM reports an error.

Example

```
.
.
.PUSH    2
LDA      .TOP,3,3
INC      .TOP,.TOP
STA      .POP,3,3
.
.
```

Reserve a number of tasks .TSK

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 can override the .TSK pseudo-op at link time by using the /TASKS= switch on the Link command line.

Example

```
.TITLE  MOD
.TSK    5       ;This program may initiate
                ;up to 5 tasks at runtime.
```

References

"Absolute Expressions" - Chapter 3
AOS/VS Link and Library File Editor (LFE) User's Manual - /TASKS= switch

Store a text string .TXT, .TXTE, .TXTF, .TXTO

Syntax

.TXT□**%string%**

.TXTE□**%string%**

.TXTF□**%string%**

.TXTO□**%string%**

Purpose

This pseudo-op directs the assembler to store the octal equivalent of an ASCII text string in consecutive memory words. In the above syntax description, **string** is an ASCII text string and **%** represents a character that you use to delimit the string. The delimiter can be any character *except*

- a character that appears in text string **string** or
- carriage return, form feed, NEW LINE, tab, space, null, or rubout

You can 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.

When the assembler encounters .TXT, it interprets the first character after the break (□) as the string 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 for one ASCII character requires seven bits of an eight-bit byte. The high-order bit, the parity bit, of each byte can be controlled by the type of .TXT pseudo-op used:

.TXT Do not explicitly set parity bit.

.TXTE Set parity bit for even parity.

.TXTF Unconditionally set parity bit.

.TXTO Set parity bit for odd parity.

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 can 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 can suppress this null storage word by using the .TXTN pseudo-op.

.TXT, .TXTE, .TXTF, .TXTO (continued)

If you want to include an absolute expression in your text string, enclose the expression in angle brackets (< >). The assembler evaluates this expression, truncates the result to 8 bits, and stores it in the appropriate byte of memory. You cannot include logical operators within the expression.

By using angle brackets, you can store the ASCII codes for characters that you could not otherwise include in your text string. For example, you cannot include a NEW LINE character in your text string. However, you can 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.

Example

```
01              00000000000             .TXTM   0       ;Pack character bytes from right
02                                                      ;to left within memory words.
03 000000       041101 020103           .TXT    "ABC D" ;These five statements store
04              000104
05 000003       041101 020103           .TXT    +ABC D+ ;the ASCII code for text
06              000104
07 000006       041101 020103           .TXT    /ABC D/ ;string 'ABC D' in memory.
08              000104
09 000011       041101 020103           .TXT    ZABC DZ ;Note the various characters we
10              000104
11 000014       041101 020103           .TXT    $ABC D$ ;use for string delimiters.
12              000104
```

See .TXTM and .TXTN for more examples.

References

"Absolute Expressions" - Chapter 3
"ASCII Character Set" - Appendix A
"Special Integer-Generating Formats" - Chapter 2
"Text Strings" - Chapter 6

Change text bytepacking .TXTM

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 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 can 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

```
01 000000    040502 041504              .TXT    "ABCDE" ;Pack bytes left/right
02           042400
03                                                  ;within words, by default.
04 000003     00000000001               (.TXTM)         ;Assemble the current value of
05                                                  ;.TXTM (1, by default).
06            00000000000              .TXTM   0       ;Pack bytes right/left.
07 000005    041101 042103              .TXT    "ABCDE"
08           000105
09 000010     00000000000               (.TXTM)         ;Assemble the current value
10                                                  ;of .TXTM.
```

References

"Absolute Expressions" - Chapter 3
"ASCII Character Set" - Appendix A
"Text Strings" - Chapter 6

Determine text string termination .TXTN

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 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 can 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.

Example

```
01 000000    030462 031464              .TXT     "1234"   ;Terminate even-length strings
02           000000
03                                                         ;with a null word, by default.
04 000003     00000000000               (.TXTN)            ;Assemble the current value of
05                                                         ;.TXTN  (0, by default).
06            00000000001               .TXTN    1         ;Do not add a null word to the
07                                                         ;end of even-length strings.
08 000005    030462 031464              .TXT     "1234"
09 000007     00000000001               (.TXTN)            ;Assemble the current value of
10                                                         ;.TXTN (1).
```

References

"Absolute Expressions" - Chapter 3
"Text Strings" - Chapter 6

Store an expression in one word and return an error if any high-order bits are set — .UWORD

Syntax

.UWORD□**expr**

Purpose

Use the .UWORD pseudo-op to generate a 16-bit storage word with the value and relocation property of **expr**.

The assembler first computes the value of the expression in 32-bit mode. Then, the assembler truncates the high-order (most significant) 16 bits and stores the resulting value in a single 16-bit word of memory.

If, before truncation, any of the original expression's high-order 16 bits are set (i.e., equal 1), the assembler returns an error. MASM generates the 16-bit storage word regardless of whether it returns an error.

The above discussion explains the Macroassembler's actions if you supply .UWORD with an absolute expression. However, if you specify a relocatable expression, MASM cannot completely resolve it, and, therefore, cannot perform the .UWORD operation. In this case, the truncation and error testing occur at link time, not assembly time.

Note that this pseudo-op does not alter the global data placement mode (set with .ENABLE).

Example

```
01 000000     00000000002              2               ;By default, each data entry
02 000002     00000000003              3               ;occupies two words of memory.
03 000004    000004                    .UWORD  4,5     ;Store the values 4 and 5
04           000005
05                                                     ;in single words of memory.
 E 000006    121543                    .UWORD  37721543
07                                                     ;The above statement generates
08                                                     ;an error because the
09                                                     ;high-order 16 bits are not
10                                                     ;all zeros.
11 000007     00000000012              12              ;The global data placement
12 000011     00000000013              13              ;mode still specifies two
13                                                     ;words per data statement.


pass 2 errors:
        EX7_48    1/5: listing   1/06: data value is out of range

1 ASSEMBLY ERROR
```

References

"Absolute Expressions" - Chapter 3
"Data Placement" - Chapter 6
"Expressions" - Chapter 2
"Relocatable Expressions" - Chapter 3

Store an expression in one word .WORD

Syntax

.WORD□**expr**

Purpose

Use the .WORD pseudo-op to generate a 16-bit storage word with the value and relocation property of **expr**. The assembler first computes the value of the expression in 32-bit mode; then truncates the leftmost 16 bits of the result.

The previous discussion explains the Macroassembler's actions if you supply .WORD with an absolute expression. However, if you specify a relocatable expression, MASM cannot completely resolve it and, therefore, cannot perform the .WORD operation. In this case, expression evaluation and truncation occur at link time, not at assembly time.

Note that this pseudo-op does not alter the global data placement mode (set with .ENABLE).

Example

```
01 000000     00000000010              10              ;MASM allocates two words for each
02 000002     00000000011              11              ;data entry, by default.
03 000004    000013                    .WORD  13,14,15 ;Store the values 13, 14, and
04           000014
05           000015
06                                                     ;15 in single words of memory.
07 000007    141762                    .WORD  37541762 ;Truncate the leftmost 16 bits
08                                                     ;and store the result (141762)
09                                                     ;in one word of memory.
10 000010     00000000021              21              ;The global data placement mode
11                                                     ;still specifies two words per
12                                                     ;data statement.
```

References

"Absolute Expressions" - Chapter 3
"Data Placement" - Chapter 6
"Expressions" - Chapter 2
"Relocatable Expressions" - Chapter 3

Delete symbol and macro definitions .XPNG

Syntax

.XPNG[<□**symbol**>]

Purpose

The .XPNG pseudo-op allows you to delete symbol and macro definitions from the current assembly. Using .XPNG, you can remove definitions for any and all symbols *except* pseudo-ops (permanent symbols). After you delete a symbol's definition, you can assign that symbol a new definition.

You can redefine numeric symbols at any time, without using .XPNG.

If you do not pass any arguments to .XPNG, the assembler removes all macro and symbol definitions, except permanent ones (pseudo-op definitions).

If you explicitly pass symbols to the .XPNG pseudo-op (in the **symbol** argument), then the assembler removes only those definitions from the current assembly. Again, you can remove any symbol, except permanent symbols (pseudo-ops), in this manner.

When MASM encounters the .XPNG pseudo-op on the first assembler pass, it immediately removes the appropriate symbol definitions. Thus, those symbol definitions are not available when MASM substitutes binary code for your source on pass two.

Since .XPNG operates on pass one, you can not use this pseudo-op to provide different definitions for a symbol during a single assembly. For example, consider the following code:

```
        ADD     0,1     ;Incorrect use of .XPNG to delete ADD.
        .
        .
        .
        .XPNG   ADD
ADD:    24
```

This code first uses the symbol ADD as an MV/Family 32-bit instruction symbol (i.e., add two accumulators). The .XPNG pseudo-op then removes symbol ADD's definition (on pass one). The subsequent source statement redefines symbol ADD as a label; i.e., the address of the memory location that contains the value 24.

.XPNG (continued)

At the beginning of pass two, the Macroassembler recognizes ADD as a label, not an instruction symbol. Thus, when MASM tries to substitute binary code for the statement

```
ADD     0,1
```

you receive an error indicating the incorrect use of symbol ADD. To avoid this problem, place the .XPNG pseudo-op at the beginning of your source module.

```
.XPNG    (continued)
```

Examples

The first example shows how .XPNG may remove specific symbol definitions.

```
01                                  .TITLE  NEWSYM
02                                  .XPNG   ADD,SUB ;Delete definitions for
03                                                  ;instruction symbols ADD
04                                                  ;and SUB.
05          00000000020             ADD:    SUB=20  ;Redefine symbols ADD and
06                                                  ;SUB as numeric symbols.
```

Our second example uses .XPNG to remove all symbol definitions, except pseudo-ops definitions.

```
01                                  .TITLE  DELALL
02                                  .XPNG           ;Delete all symbol definitions
03                                                  ;except pseudo-op definitions.
04          00000000010             ADD=10          ;Redefine symbols.
05          00000000020             SUB=20
06          00000000030             COM=30
07          00000000040             ADC=40
```

References

"Macroassembler Symbol Tables" – Chapter 8
"Symbol Tables" – Chapter 3
"Symbols" – Chapter 2

Specify lower page zero relocation .ZREL

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 can express any ZREL location in a displacement field of 8 or more bits (in any memory reference 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

```
01                                          .TITLE  Z
02                                          .ZREL           ;Place the following code in lower
03 000000 ZR  00000000010                   10              ;page zero relocatable (ZREL) memory
04 000002 ZR  00000000020                   20              ;(note the address relocation base
05                                                          ;in columns 11 and 12).
06            00000000000                   .NREL   0       ;Unshared code (NREL) memory
07 000000 UC  00000000030                   30              ;partition.
08 000002 UC  00000000040                   40
09 000004 UC  00000000050                   50
10 000006 UC  00000000060                   60
11                                          .ZREL           ;Lower page zero relocatable (ZREL)
12 000004 ZR  00000000070                   70              ;memory.  Note that MASM continues
13 000006 ZR  00000000100                   100             ;assigning ZREL addresses at the
14                                                          ;point where it left off earlier.
```

References

AOS/VS Link and Library File Editor (LFE) User's Manual
"Relocatability" - Chapter 3
"ZREL Partition" - Chapter 3

End of Chapter

8

Macroassembler Operating Procedures

This chapter provides a synopsis of procedures and conventions you need to follow to assemble, link, and execute your source code.

MASM Command Line

The CLI command line that invokes the Macroassembler is

) **XEQ MASM** *<function-switch...>* **path***<arg-switch>* *<path<arg-switch>>...* ↵

where:

XEQ	is a CLI command that executes a program.
MASM	is the name of the Macroassembler program.
function-switch	is one or more optional function switches (see Table 8-1 below).
path	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 /PASS1 (see Table 8-2 for more information on argument switches).

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 on the command line (see "Filenames" in this chapter). 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 can use two types of switches on 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 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 can appear in a single command line.

Table 8-1 describes the function switches you can 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 numeric symbols, see "Symbols" in Chapter 2.

Table 8-1 MASM Function Switches

Function Switch	Action
/16	Create a 16-bit object file that can be linked into a 16-bit process. By default, AOS/VS MASM generates a 32-bit object. This switch changes the structure of the object file and sets the default data size size to 1 word, the same as a .ENABLE WORD pseudo-op.
/$	Treat $ as a special character. By default, $ is treated like an alphabetic character. Setting this switch causes $ to be expanded instead to a number unique to each macro call.
/CPL=<n>	Set the number of characters per line on the listing and error file. <n> may be between 80 and 136, inclusive. The default number of characters per line is 80.
/E[=<filename>]	Append the error listing to <filename>. If this switch is omitted, errors are appended to @OUTPUT. If the argument to this omitted, and there is a listing file, the error file is suppressed.
/ERRORCOUNT=<n>	Terminate MASM with a fatal error if more than <n> errors occur. The default is 1024 (decimal).
/FF	If necessary to produce an even number of listing pages, append an extra form feed to the listing.
/HASH=<n>	Set the hash frame size of the symbol table. Set this switch only if /MAKEIN is also set.
/L[=<filename>]	Append a MASM listing to <filename>, or to @LIST, if <filename> is absent.
/LOCAL	Include local (nonglobal) user-defined symbols in the object file, as well as global symbols. When the /LOCAL switch is also used on Link, the local symbols can be used in debugging.
/LPP=<n>	Make each page of the listing <n> lines long. The default is 60 lines per page.
/MAKEIN	(For use of Data General MASM support personnel)
/MAKEPS	Create a permanent symbol file for use with MASM. The symbol table will have the name MASM.PS, or <filename>.PS, if you use the /PS=<filename> switch. No previous PS file is read, and no object file is created. MASM makes only one pass, and a listing may be generated.
/MULTIPLE	Report an error if symbols defined in equate statements have been previously defined with a different value in the same assembly.
/N	Do not generate an object file.
/NOPS	Do not use a PS file with this assembly.
/O=<filename>	Name the object file <filename>.OB. By default, the object file is either named after the first source file, or with the .OB pseudo-op.
/PS=<filename>	Use <filename>.PS as the PS file, rather than the default, MASM.PS.
/STATISTICS	Include elapsed and CPU time statistics in the listing (this switch requires you use the /L switch).
/SYMBOLS=<n>	Set the number of symbol name characters that MASM recognizes. The number <n> may be between 5 and 32 characters. The default is 8 characters.
/ULC	Treat uppercase and lowercase symbol names as distinct. By default, "a" and "A" are the same symbol name. Note that instruction and pseudo-op names must be given in uppercase if the /ULC switch is used.
/W	Where there is a choice, use absolute (index mode 0) addressing, instead of PC-relative (index mode 1). This switch is functionally equivalent to the .ENABLE ABS pseudo-op.

(continues)

Table 8-1 MASM Function Switches

Function Switch	Action
/XPAND	Override all listing-control pseudo-ops, including .NOCON, .NOMAC, .NOLOC and the ** flag.
/XREF={NONE, USERSYMBOLS, ALL}	Suppress or augment the listing cross-reference. /XREF=NONE suppresses the cross-reference. /XREF=USERSYMBOLS cross-references labels, macros, and user-defined numeric symbols. /XREF=ALL adds instructions and pseudo-ops to the cross-reference. The default is USERSYMBOLS (requires the /L switch).
/Z	Insert Data General proprietary header at the top of each listing page.
/8	Ignored. (See Note 1.)
/A	Ignored. (See Note 1.)
/B=<filename>	Same as /O=<filename>. (See Note 2.)
/D	Ignored. (See Note 1.)
/F	Same as /FF. (See Note 1.)
/K	Ignored. (See Note 1.)
/M	Same as /MULTIPLE. (See Note 2.)
/MEM=<n>	Ignored. (See Note 1.)
/O	Same as /XPAND. (See Note 2.)
/P	Same as /XREF=ALL. (See Note 2.)
/R	Ignored. (See Note 1.)
/S	Same as /MAKEPS. (See Note 2.)
/U	Same as /LOCAL. (See Note 2.)

(concluded)

Note 1. For compatibility with MASM16 and AOS MASM, AOS/VS MASM ignores this command-line function switch.

Note 2. For compatibility with MASM16 and AOS MASM, AOS/VS MASM recognizes this command-line function switch.

Table 8-2 describes the argument switch /PASS1. This switch can appear after any source filename in the MASM command line. For example:

```
) XEQ MASM MOD1 MOD2/PASS1 MOD3 ↵
```

Table 8-2 MASM Argument Switches

Switch	Description
<filename>/PASS1	The input file need not be read on the second pass, as it contains no code or data, only symbol and/or macro definitions.
/S	Same as /PASS1. (See Note.)

NOTE: *For compatibility with MASM16 and AOS MASM, AOS/VS MASM recognizes the /S switch.*

Linking and Executing Your Program

You cannot execute the Macroassembler output that MASM generates. You must first process your object file(s) using the Link utility. The general Link command line is

) **XEQ LINK file ...** ↵

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 on the Link command line, the .PR 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 can include a variety of switches on the command line, and you can 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/VS Link and Library File Editor (LFE) User's Manual.*

To execute a program file generated by Link, type the command

) **XEQ file** ↵

where:

file is the pathname of a program file; you need not specify the .PR extension

Filenames

Table 8-3 summarizes the AOS/VS file-naming conventions. Note that the table lists only the filename extensions, not the complete filename.

Table 8-3 AOS/VS Filename Extensions

Extension	Contents of File
.OB	Object file (generated by MASM)
.PR	Program (executable) file (generated by Link)
.PS	Permanent symbol file (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:

```
) XEQ MASM FILE1 FILE2 ↵

) XEQ MASM FILE1.SR FILE2.SR ↵
```

The object file normally receives the name of the first source file in the command line, less the .SR extension (if any) and with the .OB extension. You can specify a different name for the object file by using the /O= 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)	/O=filename	The object file receives the name you specify with the /O= 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 sample command lines will help clarify these naming conventions:

```
) XEQ MASM FILE1 FILE2 ↵
```

```
) XEQ MASM FILE1.SR FILE2.SR ↵
```

Both of the above command lines produce an object file with the name FILE1.OB.

```
) XEQ MASM/O=BOND FILE1 FILE2 ↵
```

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,

```
) XEQ MASM/O=BOND.OB FILE1 FILE2 ↵
```

also produces an object file named BOND.OB.

Permanent Symbol File

The *permanent symbol file* is a way of cutting down assembly time by segregating lengthy macros and/or assignments and assembling them ahead of time. To create a permanent symbol file, you must use the /MAKEPS switch on the assembly command line. The general MASM command for creating a permanent symbol file is

```
) XEQ MASM/MAKEPS sourcefile ... ↵
```

To create a permanent symbol file that contains definitions for all AOS/VS system calls and system parameters, use the following command:

```
) XEQ   MASM/MAKEPS PARU.32 SKIPS.SR SYSID.32 ↵
```

where:

PARU.32.SR	defines the system parameters
SKIPS.SR	contains skip macros and mnemonics for intrinsic and graphics instructions
SYSID.32.SR	contains definitions of all AOS/VS system calls

Normally, you need not issue this command line since we provide a copy of this permanent symbol file with the Macroassembler software (in file MASM.PS).

The Macroassembler places the permanent symbol file in MASM.PS, by default. If you want to place the permanent symbol file in a different file, use both the /MAKEPS and /PS= switches on the MASM command line as follows:

```
) XEQ MASM/MAKEPS/PS=filename sourcefile ... ↵
```

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 file

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 file.

Specifying a Permanent Symbol File for an Assembly

After you build a permanent symbol file, you may use that file during the assembly of your source modules. Specify a particular symbol file by using the /PS= switch in the MASM command line. The Macroassembler then uses that permanent symbol file to resolve the symbols in your source module. For example:

```
) XEQ MASM/PS=A.PS SOURCEFILE ↵
```

The Macroassembler uses file A.PS as the permanent symbol file when resolving symbols in SOURCEFILE (as described earlier in this chapter).

If you do not use the /PS= switch to specify a permanent symbol file, the Macroassembler uses file MASM.PS, if it is available.

Permanent Symbol File Size

The permanent symbol file can include up to $32{,}767_{10}$ symbols and their definitions. This figure assumes that you do not include any macro definition strings. The more macro text you place in the file, the smaller the amount of space available for symbol definitions.

Increasing symbol length does not decrease the number of symbols you can define. The following section describes symbol length in depth.

Symbol Length

The permanent symbol file specifies how many characters the Macroassembler should use to resolve the symbols in your source program. The default value for symbol length is 8 characters.

During assembly, the Macrossembler 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 8 characters):

```
LOCATION1      LOCATION2      LOCATION_START
```

To alter the symbol length for an assembly, include the /SYMBOL= switch in the MASM command line when building your permanent symbol file. Using /SYMBOL=, you may indicate any symbol length from 5 to 32 characters, inclusive. For example:

```
) XEQ    MASM/MAKEPS/PS=MASM25.PS/SYMBOL=25 SOURCE1 ↵
```

This command directs the Macroassembler to create a permanent symbol file and store it in MASM25.PS. This file will contain all the symbol definitions in file SOURCE1 and will identify those symbols according to their first 25 characters. Thus, each time you specify MASM25.PS as the permanent symbol file, MASM will resolve symbols according to their first 25 characters. For example:

```
) XEQ MASM/PS=MASM25.PS SOURCE2 ↵
```

This command directs the Macroassembler to use file MASM25.PS as the permanent symbol file. The Macroassembler will identify the symbols in source file SOURCE2 according to their first 25 characters, as specified in the permanent symbol file.

If you do not use a permanent symbol file during an assembly, the Macroassembler uses the symbol length specified in the /SYMBOL= switch, if one is present. For example:

```
) XEQ MASM/NOPS/SYMBOL=15 SOURCE3 ↵
```

The /NOPS switch instructs the Macroassembler to assemble file SOURCE3 without referring to a permanent symbol file. During this assembly, the Macroassembler resolves the symbols in SOURCE3 according to their first 15 characters.

If you use the /NOPS switch but do not include /SYMBOL=, the Macroassembler uses the first 8 characters to identify the symbols in your source code.

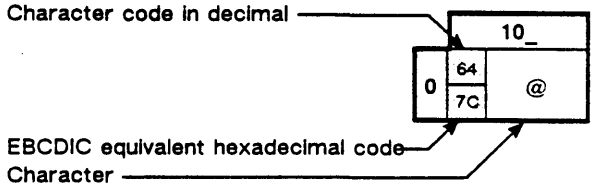
The Macroassembler ignores the /SYMBOL= switch if you do not also include either /MAKEPS or /NOPS on the MASM command line.

End of Chapter

A

ASCII Character Set

LEGEND:



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.

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)

Character code in octal at top and left of charts ↑ or ^ means CONTROL

INT-00850

End of Appendix

B

Pseudo-Op Summary

Table B-1 lists and describes the AOS/VS Macroassembler pseudo-ops. Each entry in the table indicates whether that pseudo-op can 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 assemble directives and value symbols.
- "Symbol Interpretation" in Chapter 3 explains how the Macroassembler resolves pseudo-ops that appear in your program.

Table B-1 Pseudo-Op Summary

Pseudo-op	Assembler Directive	Value Symbol	Description
.(period)		X	Current value of location counter
.ALIGN	X		Align location to a memory boundary
.ARGC, .ARGCT		X	Number of arguments passed to a macro
.ASYM	X		Define an accumulating symbol
.BLK	X		Reserve a block of data words
.CALL	X		Create data word with call relocation
.COMM	X		Define a named common symbol
.CSIZ, .CSIZE	X		Set size of unnamed common area
.DALC	X		Define an ALC (ADD) instruction
.DCMR	X		Define ELDB-format instructions
.DEMR	X		Define EJMP-format instructions
.DERA	X		Define ELEF-format instructions
.DEUR	X		Define SAVE-format instructions
.DFLM	X		Define FAMS-format instructions
.DFLS	X		Define FSST-format instructions

(continues)

Table B-1 Pseudo-Op Summary

Pseudo-op	Assembler Directive	Value Symbol	Description
.DIAC	X		Define HLV-format instructions
.DICD	X		Define ADI-format instructions
.DIMD	X		Define CIOI-format instructions
.DIMM	X		Define ADDI-format instructions
.DIMS	X		Define WSEQI-format instructions
.DIO	X		Define NIO-format instructions
.DIOA	X		Define DIA-format instructions
.DISD	X		Define PSH-format instructions
.DISS	X		Define SGT-format instructions
.DIWM	X		Define WADDI-format instructions
.DIWS	X		Define WUGTI-format instructions
.DLBA	X		Define LLEFB-format instructions
.DLBR	X		Define LPEFB-format instructions
.DLCM	X		Define LNADI-format instructions
.DLMI	X		Define LCALL-format instructions
.DLMO	X		Define LNDO-format instructions
.DLMR	X		Define LPEF-format instructions
.DLMS	X		Define LNDSZ-format instructions
.DLRA	X		Define LLEF-format instructions
.DMR	X		Define JMP-format instructions
.DMRA	X		Define LDA-format instructions
.DO	X		Repetitively assemble conditional code
.DTAC	X		Define NOVA-4 LDB-format instructions
.DUNR	X		Define WRTN and other instructions
.DUNS	X		Define DSZTS and other instructions
.DUSR	X		Define a user symbol
.DUWR	X		Define WDINC-format instructions
.DUWS	X		Define NFSSS-format instructions
.DWMM	X		Define DERR-format instructions
.DWMR	X		Define WBR-format instructions
.DWMS	X		Define WSKBO-format instructions
.DWORD	X		Create two-word data words
.DWXO	X		Define WXOP-format instruction
.DXBA	X		Define XLEFB-format instruction
.DXBR	X		Define XPEFB-format instruction
.DXCM	X		Define XNADI-format instruction
.DXMI	X		Define XCALL-format instruction
.DXMO	X		Define XNDO-format instruction
.DXMR	X		Define XPEF-format instruction
.DXMS	X		Define XNDSZ-format instruction

(continues)

Table B-1 Pseudo-Op Summary

Pseudo-op	Assembler Directive	Value Symbol	Description
.DXOP	X		Define XOP-format instruction
.DXRA	X		Define XLEF-format instruction
.EJEC, .EJECT	X		Begin a new page of the listing
.ELSE	X		Reverse sense of conditional assembly
.ENABLE	X		Control defaults
.END	X		Terminate assembly
.ENDC	X		Terminate conditional assembly
.ENT	X		Define a global entry symbol
.ENTO	X		Define an overlay entry symbol
.EOF	X		Explicit end-of-file marker
.EOT	X		Explicit end-of-file marker
.ERROR	X		Report a user-defined error message
.ESC	X		Macro escape character
.EXTC	X		Define a chain-link external
.EXTD	X		Define 8-bit address external
.EXTDA	X		Define 8-bit address external
.EXTDAN	X		Define 8-bit address of one word
.EXTDAW	X		Define 8-bit address of two words
.EXTDD	X		Define 8-bit data external
.EXTG	X		Define 32-bit data external
.EXTL	X		Define 32-bit address external
.EXTLA	X		Define 32-bit address external
.EXTLAN	X		Define 32-bit address of one word
.EXTLAW	X		Define 32-bit address of two words
.EXTLD	X		Define 32-bit data external
.EXTN	X		Define 16-bit address external
.EXTNA	X		Define 16-bit address external
.EXTNAN	X		Define 16-bit address of one word
.EXTNAW	X		Define 16-bit address of two words
.EXTND	X		Define 16-bit data external
.EXTU	X		Make all undefined symbols external
.FORC, FORCE	X		Set library force-load flag
.GADD	X		Create data word with word relocation
.GATE	X		Create gate entry with ring field
.GLOC	X		Place data relative to location
.GOTO	X		Jump ahead in conditional assembly

(continues)

Table B-1 Pseudo-Op Summary

Pseudo-op	Assembler Directive	Value Symbol	Description
.GREF	X		Create data word with gref relocation
.IFE	X		Begin conditional assembly
.IFG	X		Begin conditional assembly
.IFL	X		Begin conditional assembly
.IFN	X		Begin conditional assembly
.KCAL, .KCALL	X		Create data word with call relocation
.LCNS	X		Begin listing page with DG header
.LMIT	X		(Not supported by AOS/VS MASM)
.LOC	X	X	Place data relative to location
.LPOOL	X		Create a literal pool
.MACR, .MACRO	X		Define a macro
.MCAL, .MCALL		X	Return flag on first call to macro
.NLIT	X		Create literal pools in NREL
.NOCO, .NOCON	X	X	Suppress listing of conditional code
.NOLO, .NOLOC	X	X	Suppress listing of noncode
.NOMA, .NOMAC	X	X	Suppress listing of macro expansions
.NREL	X		Place data in relocatable NREL
.OB	X		Name object file
.PART	X		Place data in a user-defined partition
.PASS		X	Return pass number
.PENT	X		Define a procedure-entry symbol
.POP		X	Return and pop item from top of stack
.PTAR, .PTARG	X		Create data with target relocation
.PUSH	X		Push item onto top of stack
.RB	X		Name object file
.RCAL, .RCALL	X		Create data word with call relocation
.RCHA, .RCHAIN	X		Create data word with call relocation
.RDX	X	X	Radix for numeric input
.RDXO	X	X	Radix for numeric output on listing
.REV	X		Set object module revision number
.SKIP		X	Flag if previous instruction may skip
.SWORD	X		Create signed data words
.TARG	X		Create data with target relocation
.TITL, .TITLE	X		Set object module title

(continues)

Table B-1 Pseudo-Op Summary

Pseudo-op	Assembler Directive	Value Symbol	Description
.TOP		X	Return item from top of stack
.TSK	X		Set maximum concurrent task count
.TXT	X		Create data from text string
.TXTE	X		Create even-parity data from string
.TXTF	X		Create one-parity data from string
.TXTM	X	X	Set high/low packing in text strings
.TXTN	X	X	Add null byte to text strings
.TXTO	X		Create odd-parity data from string
.UWORD	X		Create unsigned data words
.WORD	X		Create one-word data words
.XPNG	X		Delete symbols from symbol table
.ZREL	X		Place data in relocatable ZREL

(concluded)

End of Appendix

C

Errors

The Macroassembler can generate two types of errors:

- command line errors
- assembly errors

Command line errors are errors you make when entering the MASM command line. *Assembly errors* are errors in your source module itself.

The following two sections of this appendix list and describe the errors in these two categories.

Command Line Errors

When you enter a MASM command line, the Macroassembler checks that it conforms to the proper syntax. If it does not, MASM returns a short message describing the error. In addition, if your command line is in error, MASM does not process your source module at all.

Table C-1 lists and describes the command line error messages.

Refer to Chapter 8 for a complete discussion on the MASM command line, including a complete list and description of all MASM command line switches.

Assembly Errors

If your MASM command line is correct, the Macroassembler assembles your source module into its binary equivalent. During this process, MASM may detect errors in your source statements. We call these *assembly errors*.

The Macroassembler always produces a listing of error messages; you cannot suppress it. By default, MASM places error messages in the generic file @OUTPUT. You can direct the error listing to a specific file by using the /E=filename switch on the MASM command line. In addition, MASM also reports errors at the end of each assembly listing. Table C-2 lists and describes the assembly error messages. Refer to "Error Listing" in Chapter 4 for more information about how MASM reports assembly errors.

Table C-1 MASM Command Line Errors

Error Message	Description
/<switchname> SWITCH REQUIRES AN ARGUMENT (See Note 1.)	The switch requires an argument, separated from it with an "=".
/<switchname>[=<argument>] IS A DUPLICATE SWITCH (See Note 1.)	The switch appears more than once on the same argument, e.g., XEQ MASM/N/N.
/<switchname>=<argument> IS AN ILLEGAL ARGUMENT VALUE (See Note 1.)	MASM reports this error if a switch expects a numeric argument and the argument was nonnumeric. It is also reported if a numeric constant is badly formed (12D4), or the value is not in the range permitted, e.g., /CPL accepts values between 80 and 136 only.
/<switchname>[=<argument>] IS AN UNKNOWN OR NON-UNIQUE SWITCH (See Note 1.)	The switch is either not known to MASM at all, such as /XYZ, or is not unique (/X for /XPAND and /XREF).
/<switchname>=<argument> SWITCH DOES NOT ACCEPT AN ARGUMENT (See Note 1.)	The switch cannot have an argument attached to it.
APPEND AND/OR WRITE ACCESS DENIED, FILE <filename> (See Notes 1 and 2.)	MASM was unable to either open or read to one of its output files, either the error, list, object or PS file.
COMMAND REQUIRES ARGUMENTS (See Note 1.)	MASM was executed without giving it the name of at least one input file on the command line.
FILE ACCESS DENIED, FILE <filename> (See Notes 1 and 2.)	MASM was able to open an input file, but was unable to read it due to its ACL (access control list).
FILE DOES NOT EXIST, FILE <filename> (See Notes 1 and 2.)	MASM could not find <filename> either with or without its assumed extension. This can occur on a source file or a PS file.
ILLEGAL FILE TYPE, FILE <filename> (See Notes 1 and 2.)	MASM was unable to read <filename> because it was not a text or data file. This usually occurs when MASM has opened a directory.
INVALID PS FILE REVISION, FILE <filename> (See Note 1.)	Before MASM uses a PS file, it checks the header revision field to be sure that it is valid. If it is not, MASM aborts with this error. NOTE: AOS/VS MASM will not accept PS files built by MASM16.
READ ACCESS DENIED, FILE <filename> (See Notes 1 and 2.)	MASM was able to open an input file, but was unable to read it due to its ACL (access control list).
WRITE ACCESS DENIED, FILE <filename> (See Notes 1 and 2.)	MASM was unable to either open or read to one of its output files, either the error, list, object or PS file.

Note 1. FATAL ERROR.

Note 2. AOS/VS SYSTEM ERROR.

Table C-2 Assembly Errors

Error Message	Description
A MACRO MAY NOT BE USED IN AN EXPRESSION	A macro symbol cannot be used as an operand in an expression.
ARGUMENT IS NOT A VALID SYMBOL NAME	A symbol name is made up of 1 to 32 characters. Certain characters are illegal in symbol names. See the section "Symbol Names" in Chapter 2.
ARGUMENT MUST BE FOLLOWED BY =	.DUSR and the instruction-defining pseudo-ops expect a symbol name, followed by an =, followed by either an instruction or an expression.
ARGUMENT VALUE IS OUT OF RANGE	The value of an instruction's argument is too large to fit in the field given. The allowable range depends on the type of instruction, the size of the field, and in some cases, the index mode.
CONFLICTING .ENABLE PSEUDO-OP ARGUMENTS	Two keyword arguments which are logical converses of each other appear in a single .ENABLE statement. An example is .ENABLE DWORD UWORD.
CONFLICTING .PART PSEUDO-OP ARGUMENTS	Two keyword arguments which are logical converses of each other appear in a single .PART statement. An example is .PART A SHORT LONG.
DATA VALUE IS OUT OF RANGE	Signed or unsigned data words, generated with either the .SWORD or .UWORD pseudo-ops, or with .ENABLE SWORD or .ENABLE UWORD in force, must fit within one 16-bit data word. For SWORD, a data word must be in the range -10000 to +77777 (octal), while for .UWORD, it must be in the range 000000 to 177777 (octal).
DUPLICATE .PART PSEUDO-OP ARGUMENTS	The same keyword argument appears more than once in a single .PART statement.
DUPLICATE .ENABLE PSEUDO-OP ARGUMENTS	The same keyword argument appears more than once in a single .ENABLE statement.
DUPLICATE <name> PSEUDO-OP	Two pseudo-ops, .OB and .REV, can appear only once per assembly.
ENTRY SYMBOL IS UNDEFINED	If a symbol is to be a .ENT, .PENT or .ASYM, its value must be defined elsewhere in the assembly.
ERROR IN A CONSTANT	A constant, decimal constant, or floating-point constant contained a numeric character not valid in the current radix, e.g. 9 when the radix is 8.
ERROR IN .TXT PSEUDO-OP ARGUMENT	The ASCII values of characters can be given within .TXT arguments by surrounding them with < and >, e.g., .TXT <377>. MASM reports an error if a < without a matching > occurs on the same line.

(continues)

Table C-2 Assembly Errors

Error Message	Description
EXPRESSION MAY NOT BE RELOCATABLE	A relocatable expression was used in a context where only an absolute expression is allowable. For example, most numeric pseudo-op arguments must be absolute.
EXPRESSION OPERAND MAY NOT BE RELOCATABLE	An operator within an expression, such as the "&" or "!" operator, expects one or both of its operands to be absolute.
EXTRANEOUS ARGUMENTS TO .TXT PSEUDO-OP	The .TXT pseudo-op accepts exactly one argument, but more than one argument was found. This error occurs frequently when the text-delimiting character is mistakenly used in the middle of a text string.
FORMAT ERROR	This error covers all other syntax errors, including an unexpected character, null arguments, etc.
ILLEGAL ALC INSTRUCTION OPTIONS	Because ALC (MOV, ADD, INC, etc.) instructions with the no load (#) option and either the no-skip or always-skip option have the same value as other instructions, MASM reports this as an error.
ILLEGAL RELOCATION	This error is reported whenever the relocation property of an expression is not allowed in the context it was found in. Examples of this include: attempting to multiply a byte- relocatable expression by 2; using a relocatable argument to an instruction where only an absolute is expected; and conversely, using an absolute expression where a relocatable one is expected.
INCORRECT NUMBER OF ARGUMENTS	All instructions expect from 1 to 4 arguments, and, depending on the type of instruction, some are optional. If too few are given, MASM assumes they are zero. If too many are given, they are ignored. This error is also reported when more than one data item is given in a single source line.
INFINITELY RECURSIVE MACRO ARGUMENT	An argument to a macro contains a reference to itself, either directly or indirectly. For example, the first argument to a macro cannot contain a reference to the first argument, e.g. ^1.
INSTRUCTION IS TOO LARGE	If an instruction's value is assigned to a symbol, e.g., .DUSR .SYST=JSR@17, the instruction must be either one or two words in size.
INSTRUCTION MAY NOT BE RELOCATABLE	When setting a symbol to an instruction's value, e.g. .DUSR .SYST=JSR@17, the instruction's arguments cannot be a label or any other relocatable expression.

(continues)

Table C-2 Assembly Errors

Error Message	Description
INSUFFICIENT MEMORY AVAILABLE, FILE <filename> (See Notes 1 and 2.)	MASM allocates memory dynamically for many of its internal databases. If this memory is exhausted, MASM aborts with this error. This error is almost always caused by an infinite recursion of macro calls, e.g., a macro unconditionally calling itself.
INTERNAL EXPRESSION CONSISTENCY ERROR (See Note 3.)	Some problem exists in the internal representation of an expression. Correct all other errors (if any) first.
INVALID USE OF INDIRECTION	The indirection indicator (@) can be used only with an instruction which can be indirected.
INVALID USE OF NOLOAD OPTION	The no-load indicator (#) can be used only with the ALC (MOV, ADD, SUB, etc.) instructions.
INVALID USE OF .ELSE PSEUDO-OP	A .ELSE pseudo-op can occur only if a properly nested .IFx pseudo-op appeared prior to it. A .ELSE cannot terminate a .DO conditional loop, nor can it be the target of a .GOTO pseudo-op.
INVALID USE OF .ENDC PSEUDO-OP	A .ENDC pseudo-op can occur only if a properly nested .IFx or .DO pseudo-op appeared prior to it. A .ENDC cannot be the target of a .GOTO pseudo-op.
INVALID .NREL PSEUDO-OP ARGUMENT VALUE	The only allowable values for the argument to .NREL in AOS/VS MASM are: 0, 1, 2, 4, 5, 6 or 7.
LABEL IS MULTIPLY DEFINED	A new definition of a symbol is in conflict with a previous definition. The previous definition may be in the same source, in the .PS file, or in MASM's permanent table of instructions and pseudo-ops.
LABEL IS NOT A VALID SYMBOL NAME	A symbol name is made up of 1 to 32 characters, the first being alphabetic, the rest being either alphabetic or numeric.
LABEL VALUE PHASE ERROR LITERAL POOL PHASE ERROR	Because AOS/VS MASM makes two passes over the source, it checks that the relocatable value of labels are the same from pass to pass, and reports an error if they are not. This is usually due to omission or inclusion of code or data on the second pass and not on the first. This can happen either from the use of the /PASS1 command-line switch or the .PASS pseudo-op.
LITERALS MAY NOT BE USED PRIOR TO .NLIT PSEUDO-OP	The .NLIT pseudo-op affects placement of literal pools, and so it must occur prior to any literal use in the source. .NLIT is normally placed at the beginning of a source file.

(continues)

Table C-2 Assembly Errors

Error Message	Description
UNKNOWN .PART PSEUDO-OP ARGUMENT	The pseudo-op, .PART, accepts additional keyword arguments controlling its operation. The arguments must be in uppercase if case-sensitive assembly is being done.
UNKNOWN .ENABLE PSEUDO-OP ARGUMENT	The pseudo-op, .ENABLE, accepts additional keyword arguments controlling its operation. The arguments must be in uppercase if case-sensitive assembly is being done.
MISMATCHED PARENTHESIS IN AN EXPRESSION	Parentheses must be properly nested and balanced in expressions. Space, tab or any other break character cannot appear in the middle of an expression.
OPERAND OF THE B OPERATOR IS OUT OF RANGE	The second operand of the shift operator, B, must be within its appropriate range (must be between 0 and 15 decimal).
OPERAND OF THE S OPERATOR IS OUT OF RANGE	The second operand of the shift operator, S, must be within its appropriate range (must be between 0 and 31 decimal).
PARTITION ATTRIBUTES CONFLICT WITH PREVIOUSLY DEFINED ATTRIBUTES	The partition attribute keywords given on one partition definition differ with those given on another definition of the same partition in the same assembly. Some attributes can be resolved without error, e.g., SHORT and LONG, but MASM reports an error for SHARED-UNSHARED and DATA-CODE attribute conflicts.
POSSIBLE SKIP AT END OF PARTITION POSSIBLE SKIP INTO DATA POSSIBLE SKIP INTO INSTRUCTION POSSIBLE SKIP INTO TEXT	MASM generally knows when an instruction could possibly attempt to skip the word following the instruction. If the word directly following the instruction is not a one-word data item, MASM reports that an instruction is skipping either into a two-, three-, or four-word instruction, into a two-word data item, or into a .TXT pseudo-op. This error is also reported when the last instruction in a partition skips.
STACK IS EMPTY	Either the .POP and/or .TOP value pseudo-op was used at a point in the assembly process when no elements were pushed onto the stack.
SYMBOL IS MULTIPLY DEFINED	A new definition of a symbol is in conflict with a previous definition. The previous definition may be in the same source, in the .PS file, or in MASM's permanent table of instructions and pseudo-ops.

(continues)

Table C-2 Assembly Errors

Error Message	Description
SYMBOL IS UNDEFINED	An expression refers to a symbol which is not defined in this module, the PS file, or the permanent table.
SYMBOL MAY NOT BE AN ENTRY	A macro, external or permanent instruction symbol cannot be a .ENT, .PENT or .ASYM.
SYMBOL MUST BE AN EXTERNAL	The .CALL, .TARG and other pseudo-ops accept as an argument a symbol, and that symbol must be an external for the call to be resolved correctly.
SYMBOL TYPE CONFLICT	A symbol cannot be an external and an entry, or an external and macro, etc.
SYNTAX ERROR IN A CONSTANT	A constant, decimal constant or floating-point constant contained an unexpected character, for example, 45Z4.
SYNTAX ERROR IN AN EXPRESSION	An expression was not formed correctly. An operand may be missing (4++5), or an operator may be missing (LDA 4+,3). Note that MASM expressions cannot contain spaces, e.g., 4 + 5 is not legal.
SYMBOL TABLE IS FULL, FILE <filename> (See Note 1.)	As of this writing, MASM's symbol table has room for approximately 32,767 symbols. If this limit is exceeded, MASM aborts with this error. Note that macro symbol definitions use significantly more symbol table space than other symbols. Eliminating unnecessary text from macros may eliminate symbol table overflow.
TOO MANY ARGUMENTS TO A MACRO	A macro cannot be called with more than 64 arguments.
UNTERMINATED CONDITIONAL	A conditional part of an assembly begun with a .DO or .IFx must be explicitly terminated with a .ENDC before the end of either the macro or the file that the conditional assembly began in.
UNTERMINATED MACRO CALL	Actual macro arguments on a macro call can be passed inside of square brackets, and can extend over several lines. However, if a [is seen but no] is seen by the end of the input file, MASM reports an error.
UNTERMINATED .TXT PSEUDO-OP ARGUMENT	The argument to .TXT must be a text string which must begin and end with the same character. MASM reports an error if it reaches the end of the input file without finding a closing text delimiter.

(continues)

Table C-2 Assembly Errors

Error Message	Description
USER ERROR	This error is reported whenever the .ERROR pseudo-op is seen.
.XPNG OF A PSEUDO-OP	.XPNG can remove instructions, but not pseudo-ops, from the permanent table of instructions and pseudo-ops.
<name> PSEUDO-OP ARGUMENT VALUE IS OUT OF RANGE	Arguments to .REV and .TSK must be in the range 0-255 (decimal) for MASM to format them correctly in the object file. The .RDX pseudo-op accepts 2-20 (decimal), while .RDXO accepts 8-20 (decimal). .ALIGN and .PART's ALIGN= attribute accept 0-10 (decimal).
<name> PSEUDO-OP ARGUMENT VALUE MUST BE NON-NEGATIVE	The numeric arguments to the .BLK, .COMM, .CSIZE, and .DO pseudo-ops must be zero or greater.
<name> PSEUDO-OP DOES NOT ACCEPT ARGUMENTS	Many pseudo-ops do not accept arguments, e.g., no text may appear after them on the line (except for comments).
<name> PSEUDO-OP MAY NOT BE USED IN AN EXPRESSION	Some pseudo-ops return values when used in an expression, e.g., .MCALL or .RDX. Others, like .MACRO or .NREL, do not.
<name> PSEUDO-OP REQUIRES ARGUMENT(S)	Many pseudo-ops, including .ENABLE, .ENTO, .REV, .PART, and .COMM require at least one argument, and will accept more.
.LMIT PSEUDO-OP IS NOT SUPPORTED	The .LMIT pseudo-op is permitted with MASM16 and some other Data General macroassemblers, but not with AOS/VS MASM.

(concluded)

Note 1. *FATAL ERROR.*

Note 2. *AOS/VS SYSTEM ERROR.*

Note 3. *MASM INTERNAL ERROR. If this error persists, and there are no other user errors, please report it to your system manager or the appropriate Data General MASM support personnel.*

End of Appendix

D

Compatibility between AOS MASM, AOS/VS MASM16, and AOS/VS MASM

In general, AOS/VS MASM is upward-compatible from AOS MASM and AOS/VS MASM16. AOS/VS MASM can usually assemble any assembler source that the 16-bit assemblers can. The opposite is very often not true. The following are known inconsistencies in upward-compatibility from AOS MASM and AOS/VS MASM16 to AOS/VS MASM:

- Command-line changes: AOS/VS MASM recognizes both its switches and AOS MASM's. However, some items may require additional switches for compatibility with AOS MASM. These switches are:

/16	Forces generation of a 16-bit object file, and makes the size of data words 16 bits.
/$	Causes the $ character to expand to a macro invocation number.
/SYMBOL=5	Since AOS MASM's default symbol length is five characters, and AOS/VS MASM's default is eight. Omit this switch if the AOS MASM /8 switch was used.

- Syntax of instructions using literals: When literals are used in AOS/VS MASM, commas must be placed between instruction arguments. AOS MASM does not require this. For example, "LDA 1,=2" is acceptable to both, but only AOS/VS MASM will not accept "LDA 1 =2".

- Non-line-replacement macros: AOS MASM permits one-line macros to be defined and used in the middle of statements, e.g., ".TITLERUNT [SYST] [REV]", where SYST and REV are macro names. AOS/VS MASM does not permit this and will report a format error.

- Macro termination character placement: The "%" that terminates a macro definition must be the first character on the line in AOS/VS MASM. This restriction does not exist in AOS MASM.

- Continued macros: AOS MASM permits a macro definition to be closed, then restarted and additional text added. AOS/VS MASM considers this as a multiple symbol definition error.

- Spelling errors in macro and pseudo-op names: Because AOS MASM uses only the first five characters of macro and pseudo-op names, even when the /8 switch is set, it would not flag spelling errors in the remaining characters. AOS/VS MASM will report an error if, for instance, ".NOCON" is misspelled as ".NOCOC".

- "_" in constants to resolve "B," "D," and "E" ambiguities: When using radices greater than 10, the characters "B", "D" and "E" could be either digits or operators. To resolve this ambiguity, AOS MASM permits an "_" to precede the character if it is to be interpreted as an operator. AOS/VS MASM will not recognize this and will report it as an error.

- .LMIT pseudo-op: AOS/VS MASM will report that .LMIT is unimplemented.

- .PUSH .LOC--.LOC .POP: AOS MASM recognizes a special case in .PUSH .LOC, followed at some point by a .LOC .POP. The location is reset to the *end* of the pushed partition, not the actual location pushed. AOS/VS MASM does not do this.

- Conflicts with permanent symbols: AOS/VS MASM has many more pseudo-ops than AOS MASM, and in addition has the instruction set permanently built into it. Sometimes these permanent symbol names will conflict with user symbol name definitions, and AOS/VS MASM will report an error.

- Miscellaneous syntax errors: AOS/VS MASM will catch errors that AOS MASM will miss, such as "LDA 1.TEMP" or "LDA 1,,TEMP,3".

- Unterminated conditionals: AOS MASM will automatically close any conditional-assembly blocks at the end of a macro. AOS/VS MASM will report an error.

- Symbol name length may not be changed at assembly time: AOS MASM permits a five-character PS file to be used in an eight-character assembly and vice-versa. AOS/VS MASM permits the name length to be set only when the PS file is built.

- Default index mode sometimes different: AOS MASM will generate index mode 0 in a reference from .NREL 1 to .NREL 0, by default. AOS/VS MASM will generate index mode 1 by default. In other cases, the default will be the same.

- "." and labels always relocatable: AOS MASM makes "." and labels relocatable if they are part of ZREL or NREL, and absolute if they are in absolute. Because of the 32-bit ring architecture, AOS/VS MASM always makes them relocatable, even in absolute. This affects where symbols can be used in expressions.

- Some relocatable expressions not permitted: AOS MASM allows some expressions involving relocatable operands which AOS/VS MASM does not. "X/2", where X is byte-relocatable, is an example. This also is the case with the relational operators and relocatable symbols.

- Some expressions inside .TXT arguments: AOS MASM will assemble .TXT '<<">>' correctly. AOS/VS MASM will not.

- T' undocumented operator: T' before a symbol name in AOS MASM gets relocation information on the symbol. AOS/VS MASM does not support this and will report an error.

- Single- vs. double-precision internal results: AOS MASM uses single-precision in expression evaluation while AOS/VS MASM uses double-precision. As a result, 177777==-1 is true in AOS MASM, false in AOS/VS MASM.

- PC-relative with an explicit index of 1: If a memory-reference instruction has a relocatable displacement and an explicit index of 1, AOS/VS MASM will automatically subtract "." from the displacement. AOS MASM will not. However, AOS MASM sources which use "EJMP (X-.)&177777,1" to accomplish the same result will still assemble correctly.

- Range of ELEF arguments: When using ELEF to load a constant, the constant must be a 15-bit unsigned number (e.g., between 0 and 32767). AOS MASM would not check this. AOS/VS MASM will check this and report an error.

- Literal pool differences: AOS/VS MASM puts items in literal pools in the reverse order from AOS MASM.

- PS files: PS files built by AOS MASM are not usable with AOS/VS MASM.

- Object files: Object files may or may not match word-for-word, even if they are functionally equivalent.

- Listing files: Listing files will not be the same.

- .NREL argument values: AOS MASM accepts any nonzero .NREL argument as equivalent to ".NREL 1". AOS/VS MASM allows only 0, 1, 2, 4, 5, 6 and 7 as an argument to .NREL, and assumes particular meaning for each.

End of Appendix

Index

Within the index, the letter "f" following a page entry indicates "and the following page"; the letters "ff" following a page entry indicate "and the following pages."

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

DataGeneral

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 _____

Data General Corporation
Educational Services/TIPS
MS G214
2400 Computer Drive
Westboro, MA 01580
(617) 366-8911, Extension 1610

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

012-1780

educational services

CUT ALONG DOTTED LINE

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.

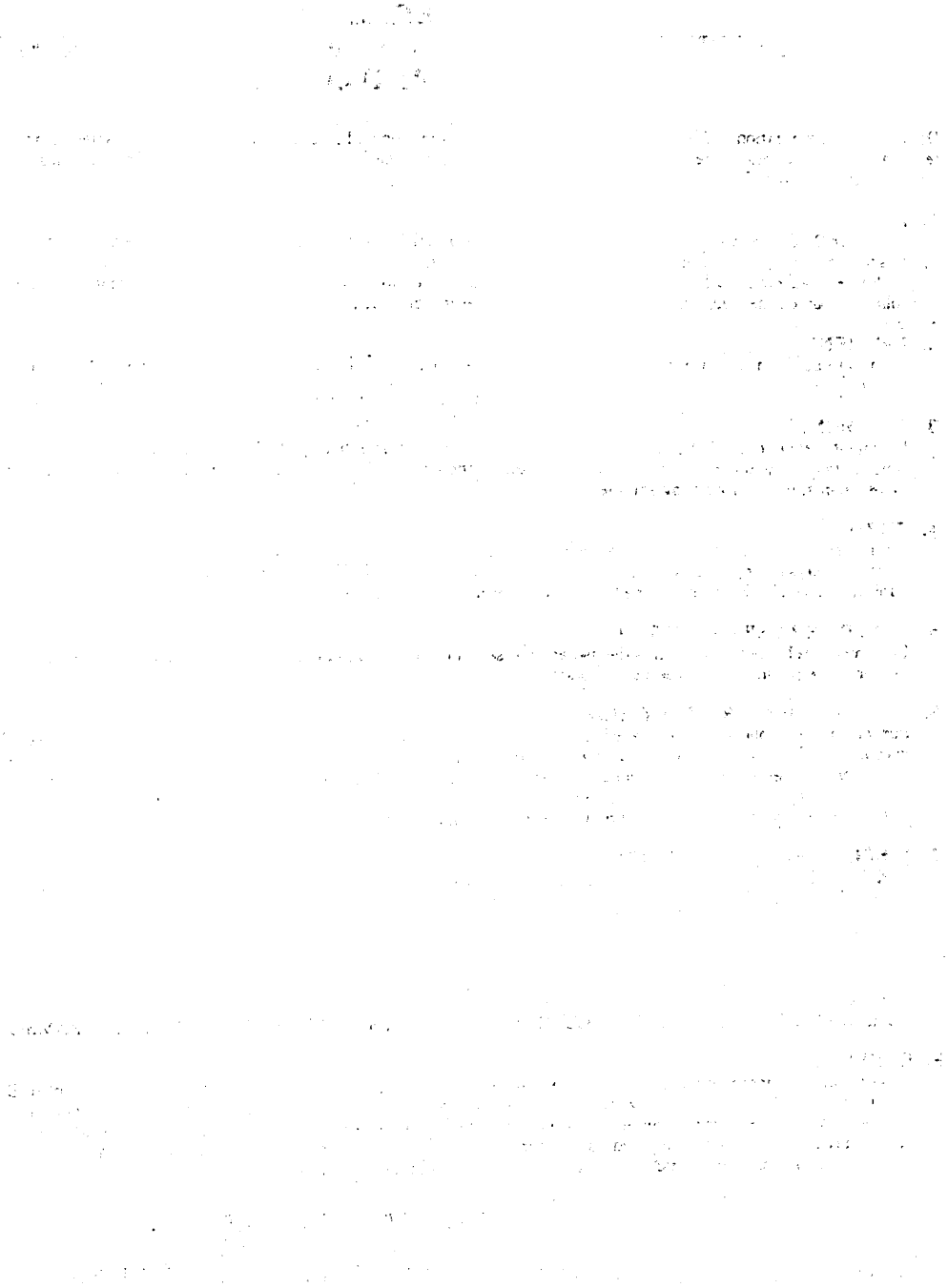
DISCOUNT SCHEDULES

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.



093-000242-02

Data General Corporation, Westboro, MA 01580