digital

VAX–11 MACRO
Language Reference Manual

Order No. AA-D032C-TE

VAX11

March 1980

This document describes the assembly language supported by VAX/VMS. All symbols, expressions, addressing modes, and directives are detailed. No prior knowledge of the VAX-11 MACRO assembler is assumed.

VAX–11 MACRO

Language Reference Manual

Order No. AA-D032C-TE

SUPERSESSION/UPDATE INFORMATION:	This revised document supersedes the VAX-11 MACRO Language Reference Manual (Order No. AA-D032B-TE)
OPERATING SYSTEM AND VERSION:	VAX/VMS V02
SOFTWARE VERSION:	VAX-11 MACRO V02

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, August 1978
Revised, February 1979
Revised, March 1980

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

CONTENTS

			Page
PREFACE			vii
CHAPTER	1	INTRODUCTION	1-1
CHAPTER	2	MACRO SOURCE STATEMENT FORMAT	2-1
	2.1	LABEL FIELD	2-2
	2.2	OPERATOR FIELD	2-3
	2.3	OPERAND FIELD	2-3
	2.4	COMMENT FIELD	2-4
CHAPTER	3	THE COMPONENTS OF MACRO SOURCE STATEMENTS	3-1
	3.1	CHARACTER SET	3-1
	3.2	NUMBERS	3-3
	3.2.1	Integers	3-3
	3.2.2	Floating-Point Numbers	3-4
	3.2.3	Packed Decimal Strings	3-5
	3.3	SYMBOLS	3-5
	3.3.1	Permanent Symbols	3-5
	3.3.2	User-defined Symbols and Macro Names	3-6
	3.3.3	Determining Symbol Values	3-6
	3.4	LOCAL LABELS	3-7
	3.5	TERMS AND EXPRESSIONS	3-9
	3.6	UNARY OPERATORS	3-11
	3.6.1	Radix Control Operators	3-12
	3.6.2	Textual Operators	3-13
	3.6.2.1	ASCII Operator	3-13
	3.6.2.2	Register Mask Operator	3-13
	3.6.3	Numeric Control Operators	3-14
	3.6.3.1	Floating-Point Operator	3-14
	3.6.3.2	Complement Operator	3-15
	3.7	BINARY OPERATORS	3-15
	3.7.1	Arithmetic Shift Operator	3-16
	3.7.2	Logical AND Operator	3-16
	3.7.3	Logical Inclusive OR Operator	3-17
	3.7.4	Logical Exclusive OR Operator	3-17
	3.8	DIRECT ASSIGNMENT STATEMENTS	3-17
	3.9	CURRENT LOCATION COUNTER	3-18
CHAPTER	4	ADDRESSING MODES	4-1
	4.1	GENERAL REGISTER MODES	4-1
	4.1.1	Register Mode	4-4
	4.1.2	Register Deferred Mode	4-4
	4.1.3	Autoincrement Mode	4-5
	4.1.4	Autoincrement Deferred Mode	4-5
	4.1.5	Autodecrement Mode	4-6
	4.1.6	Displacement Mode	4-6
	4.1.7	Displacement Deferred Mode	4-8
	4.1.8	Literal Mode	4-9

CONTENTS

			Page
	4.2	PROGRAM COUNTER MODES	4-10
	4.2.1	Relative Mode	4-10
	4.2.2	Relative Deferred Mode	4-11
	4.2.3	Absolute Mode	4-12
	4.2.4	Immediate Mode	4-12
	4.2.5	General Mode	4-13
	4.3	INDEX MODE	4-14
	4.4	BRANCH MODE	4-17
CHAPTER	5	GENERAL ASSEMBLER DIRECTIVES	5-1
		.ADDRESS	5-3
		.ALIGN	5-4
		.ASCIx	5-6
		.ASCII	5-7
		.ASCIC	5-7
		.ASCID	5-8
		.ASCIZ	5-8
		.BLKx	5-9
		.BYTE	5-11
		.CROSS .NOCROSS	5-12
		.DEBUG	5-14
		.DEFAULT	5-15
		.D_FLOATING .DOUBLE	5-16
		.DISABLE	5-17
		.ENABLE	5-17
		.END	5-20
		.ENDC	5-20
		.ENTRY	5-21
		.ERROR	5-23
		.EVEN	5-24
		.EXTERNAL	5-24
		.F_FLOATING .FLOAT	5-25
		.G_FLOATING	5-26
		.GLOBAL	5-27
		.H_FLOATING	5-28
		.IDENT	5-29
		.IF	5-30
		.IF_x	5-33
		.IIF	5-36
		.LIST	5-37
		.LONG	5-38
		.MASK	5-39
		.NLIST	5-39
		.NOCROSS	5-40
		.NOSHOW	5-40
		.OCTA	5-41
		.ODD	5-41
		.OPDEF	5-42
		.PACKED	5-44
		.PAGE	5-44
		.PRINT	5-45
		.PSECT	5-46
		.QUAD	5-51
		.REFn	5-52
		.RESTORE_PSECT	5-53
		.SAVE_PSECT	5-54
		.SHOW .NOSHOW	5-55

CONTENTS

			Page
		.SIGNED_BYTE	5-57
		.SIGNED_WORD	5-58
		.SUBTITLE	5-60
		.TITLE	5-61
		.TRANSFER	5-62
		.WARN	5-65
		.WEAK	5-66
		.WORD	5-67
CHAPTER	6	MACROS	6-1
	6.1	ARGUMENTS IN MACROS	6-2
	6.1.1	Default Values	6-3
	6.1.2	Keyword Arguments	6-3
	6.1.3	String Arguments	6-4
	6.1.4	Argument Concatenation	6-6
	6.1.5	Passing Numeric Values of Symbols	6-7
	6.1.6	Created Local Labels	6-7
	6.1.7	Macro String Operators	6-8
	6.1.7.1	%LENGTH Operator	6-9
	6.1.7.2	%LOCATE Operator	6-10
	6.1.7.3	%EXTRACT Operator	6-11
	6.2	MACRO DIRECTIVES	6-13
		.ENDM	6-13
		.ENDR	6-13
		.IRP	6-14
		.IRPC	6-16
		.LIBRARY	6-18
		.MACRO	6-19
		.MCALL	6-21
		.MDELETE	6-22
		.MEXIT	6-23
		.NARG	6-24
		.NCHR	6-25
		.NTYPE	6-26
		.REPEAT	6-28
APPENDIX	A	ASCII CHARACTER SET	A-1
APPENDIX	B	VAX-11 MACRO ASSEMBLER DIRECTIVES AND LANGUAGE SUMMARY	B-1
	B.1	ASSEMBLER DIRECTIVES	B-1
	B.2	SPECIAL CHARACTERS	B-7
	B.3	OPERATORS	B-9
	B.3.1	Unary Operators	B-9
	B.3.2	Binary Operators	B-10
	B.3.3	Macro String Operators	B-10
	B.4	ADDRESSING MODES	B-11
APPENDIX	C	PERMANENT SYMBOL TABLE	C-1
	C.1	OPCODES (ALPHABETIC ORDER)	C-1
	C.2	OPCODES (NUMERIC ORDER)	C-8

CONTENTS

			Page
APPENDIX	D	HEXADECIMAL/DECIMAL CONVERSION	D-1
	D.1	HEXADECIMAL TO DECIMAL	D-1
	D.2	DECIMAL TO HEXADECIMAL	D-1
	D.3	POWERS OF 2 AND 16	D-2
INDEX			Index-1

FIGURES

FIGURE	5-1	Using Transfer Vectors	5-63

TABLES

TABLE	3-1	Special Characters Used in VAX-11 MACRO Statements	3-1
	3-2	Separating Characters in VAX-11 MACRO Statements	3-3
	3-3	Unary Operators	3-11
	3-4	Binary Operators	3-15
	4-1	Addressing Modes	4-2
	4-2	Floating Point Short Literals	4-9
	4-3	Index Mode Addressing	4-16
	5-1	Summary of General Assembler Directives	5-1
	5-2	.ENABLE and .DISABLE Symbolic Arguments	5-18
	5-3	Condition Tests for Conditional Assembly Directives	5-31
	5-4	Operand Descriptors	5-43
	5-5	Program Section Attributes	5-48
	5-6	Default Program Section Attributes	5-49
	5-7	.SHOW and .NOSHOW Symbolic Arguments	5-56
	6-1	Summary of Macro Directives	6-2
	A-1	Hexadecimal/ASCII Conversion	A-1
	B-1	Assembler Directives	B-1
	B-2	Special Characters Used in VAX-11 MACRO Statements	B-7
	B-3	Unary Operators	B-9
	B-4	Binary Operators	B-10
	B-5	Macro String Operators	B-10
	B-6	Addressing Modes	B-11
	D-1	Hexadecimal/Decimal Conversion	D-3

PREFACE

MANUAL OBJECTIVES

This manual describes the VAX-11 MACRO language. It includes the format and function of each feature of the language. The VAX-11 MACRO User's Guide describes how to use VAX-11 MACRO.

INTENDED AUDIENCE

This manual is intended for all programmers writing VAX-11 MACRO programs. Programmers should be familiar with assembly language programming, the VAX-11 instruction set, and the VAX/VMS operating system before reading this manual.

The VAX-11 MACRO User's Guide provides a brief introduction to the assembler and describes the commands necessary to use VAX-11 MACRO. The VAX-11 Architecture Handbook describes the VAX-11 instruction set. All programmers should read these manuals before using this language reference manual.

STRUCTURE OF THIS DOCUMENT

This manual is organized into six chapters and four appendixes, as follows:

- Chapter 1 introduces the features of the VAX-11 MACRO language.
- Chapter 2 describes the format used in VAX-11 MACRO source statements.
- Chapter 3 describes the components of VAX-11 MACRO source statements: the character set; numbers; symbols; local labels; terms and expressions; unary and binary operators; direct assignment statements; and the current location counter.
- Chapter 4 summarizes and gives examples of the use of the VAX-11 MACRO addressing modes.
- Chapter 5 describes the VAX-11 MACRO general assembler directives.

- Chapter 6 describes the directives used in defining and expanding macros.
- Appendix A lists the ASCII character set that can be used in VAX-11 MACRO programs.
- Appendix B summarizes the general assembler and macro directives (in alphabetical order), special characters, unary operators, binary operators, and addressing modes.
- Appendix C lists alphabetically and by opcode the permanent symbols (instruction set) defined for use with VAX-11 MACRO.
- Appendix D gives rules for hexadecimal/decimal conversion.

ASSOCIATED DOCUMENTS

The following documents are relevant to VAX-11 MACRO programming:

- <u>VAX-11 Architecture Handbook</u>
- <u>VAX-11 MACRO User's Guide</u>
- <u>VAX/VMS Command Language User's Guide</u>
- <u>VAX-11 Linker Reference Manual</u>
- <u>VAX-11 Symbolic Debugger Reference Manual</u>

For a complete list of all VAX-11 documents, including a brief description of each, see the <u>VAX-11 Information Directory and Index.</u>

CONVENTIONS USED IN THIS DOCUMENT

The following conventions are observed in this manual, as in the other VAX-11 documents:

Convention	Meaning
Uppercase words and letters	Uppercase words and letters, used in examples, indicate that you should type the word or letter exactly as shown.
Lowercase words and letters	Lowercase words and letters, used in format examples, indicate that you are to substitute a word or value of your choice.
[ ]	Square brackets indicate that the enclosed item is optional.
...	A horizontal ellipsis indicates that the preceding item(s) can be repeated one or more times.
. . .	A vertical ellipsis indicates that not all of the statements in an example or figure are shown.

SUMMARY OF TECHNICAL CHANGES

This manual documents VAX-11 MACRO V2.0, as released with Release 2.0 of VAX/VMS. This section summarizes the technical changes from the last version, released with Release 1.5 of VAX/VMS.

Changes affecting directives:

There is a new data type, the octaword, which stores 128 bits (16 bytes) of binary data. The octaword storage directive is .OCTA. A new block storage directive, .BLKO, reserves octawords for data. A new operand generation directive, .REF16, generates an octaword operand. However, DIGITAL recommends using the opcode definition directive, .OPDEF, instead of .REF; there are new operand descriptors for use with .OPDEF which describe octaword operators.

Two new floating-point directives provide additional flexibility and capacity for floating-point numbers. The .G_FLOATING directive generates 64-bit data (quadwords) of which one bit is the sign, 11 bits are exponent, and 52 bits are fraction. The .H_FLOATING directive generates 128-bit data (octawords) with one bit for sign, 15 bits for exponent, and 112 bits for fraction. There are new block storage directives, .BLKG and .BLKH, which reserve G_floating quadwords and H_floating octawords, respectively. There are new operand descriptors for use with .OPDEF which describe G_floating and H_floating operands. (Note: not all VAX-11 processors support the G_floating and H_floating data types.)

To bring the formats of the floating-point data types and directives into a consistent format, there are new alternate forms of the .FLOAT directive — .F_FLOATING — and the .DOUBLE directive — .D_FLOATING. The terms F_floating and D_floating are used in this manual to denote single-precision and 64-bit double precision data.

A new program section attribute, VEC, has been added for use with the .PSECT directive. It shows that a program section contains information to be used in a privileged shareable image.

The new directives, attributes, and operand descriptors listed above are all described under the directive names in Chapter 5.

New opcodes have been added to the VAX-11 instruction set to make use of octaword, G_floating, and H_floating data. These opcodes, and the mnemonics that represent them in source code, are listed in Appendix C.

Miscellaneous changes:

- User-defined symbols may now be up to 31 characters long.
- The ASCII string used with the ASCII operator (^A) may now be up to 16 characters long, depending on the data type of the operand.
- Source statements used within macros may now be up to 1000 characters long.

CHAPTER 1

INTRODUCTION

The VAX-11 MACRO programming language is an assembly language for programming VAX-11 computers under the VAX/VMS operating system. Source programs written in the VAX-11 MACRO programming language are translated into object (or binary) code by the VAX-11 MACRO assembler, which produces an object module and, optionally, a listing file. These functions of the assembler are described in the VAX-11 MACRO User's Guide. The features of the language itself are introduced in this chapter.

VAX-11 MACRO source programs consist of a sequence of source statements. These source statements may be any of the following:

- VAX-11 native-mode instructions
- Direct assignment statements
- Assembler directives

Instructions manipulate data. They perform such functions as addition, data conversion, and transfer of control. Instructions are usually followed in the source statement by operands, which can be any kind of data needed for the operation of the instruction. The VAX-11 instruction set is summarized in Appendix C of this volume and is described in detail in the VAX-11 Architecture Handbook.

Direct assignment statements equate symbols to values.

Assembler directives guide the assembly process and provide tools for using the instructions. There are two classes of assembler directives: general assembler directives and macro directives.

General assembler directives can be used to perform the following:

- Store data or reserve memory for data storage
- Control the alignment in memory of parts of the program
- Specify the methods of accessing the sections of memory in which the program will be stored.
- Specify the entry point of the program or of part of the program
- Specify the way in which symbols will be referenced
- Specify that a part of the program is to be assembled only under certain conditions

- Control the format and content of the listing file
- Display informational messages
- Control the assembler options that are used to interpret the source program
- Define new opcodes

Macro directives are used to define macros and repeat blocks. They allow programmers to do the following:

- Repeat identical or similar sequences of source statements throughout a program
- Use string operators to manipulate and test the contents of source statements

Use of macros and repeat blocks helps to minimize programmer errors and to speed the debugging process.

CHAPTER 2

MACRO SOURCE STATEMENT FORMAT

A source program consists of a sequence of source statements, which the assembler interprets and processes, one by one, generating object code or performing a specific assembly-time process. A source statement can occupy one source line or can extend onto several source lines. Each source line can be up to 132 characters long; however, no line should exceed 80 characters to ensure that the source line fits (with its binary expansion) on one line in the listing file.

MACRO statements can consist of up to four fields:

- Label field -- symbolically defines a location in a program
- Operator field -- specifies the action to be performed by the statement; this field can be an instruction, an assembler directive, or a macro call
- Operand field -- contains the instruction operand(s) or the assembler directive argument(s) or the macro argument(s)
- Comment field -- contains a comment that explains the meaning of the statement; this field does not affect program execution

The label field and the comment field are optional. The label field ends with a colon (:) and the comment field starts with a semicolon (;). The operand field must conform to the format of the instruction, directive, or macro specified in the operator field.

Although statement fields can be separated by either a space or a tab (see Table 3-2), formatting statements with the tab character is recommended for consistency and clarity. By DIGITAL convention, tab characters are used to separate the statement fields as follows:

Field	**Begins in Column**	**Tab Characters to Reach Column**
Label	1	0
Operator	9	1
Operand	17	2
Comment	41	5

For example:

```
        .TITLE  ROUT1
        .ENTRY  START,0                 ; BEGINNING OF ROUTINE
        CLRL    R0                      ; CLEAR REGISTER
LABT:   SUBL3   #10,4(AP)R2             ; SUBTRACT 10
LAB2:   BRB     CONT                    ; BRANCH TO ANOTHER ROUTINE
```

A single statement can be continued on several lines by using a hyphen (-) as the last nonblank character before the comment field or at the end of line (when there is no comment). For example:

```
LAB1:   MOVAL   W^BOO$AL_VECTOR,-     ; SAVE ADDRESS OF
                RPB$L_IOVEC(R7)       ; BOOT DEVICE DRIVER.
```

VAX-11 MACRO treats the above statement as equivalent to the following statement:

```
LAB1:   MOVAL   W^BOO$AL_VECTOR,RPB$L_IOVEC(R7) ; SAVE BOOT DRIVER
```

A statement can be continued at any point. User-defined and permanent symbol names; however, should not be continued on two lines. If a symbol name is continued and the first character on the second line is a tab or a blank, the symbol name will be terminated at that character. (Section 3.3 describes symbols in detail.)

Note that when a statement occurs in a macro definition (see Chapter 6), the statement cannot contain more than 1000 characters.

Blank lines, although legal, have no significance in the source program except that they terminate a continued line.

The following sections describe each of the statement fields in detail.

2.1 LABEL FIELD

A label is a user-defined symbol that identifies a location in the program. The symbol is assigned a value equal to the location counter at the location in the program section in which the label occurs (see the VAX-11 MACRO User's Guide for information on program sections). The user-defined symbol name can be up to 31 characters long and can contain any alphanumeric character and the underline(_), dollar sign ($), and period (.) characters. Section 3.3 describes the rules for forming user-defined symbol names in more detail.

If a statement contains a label, the label must be in the first field on the line.

A label is terminated by a colon (:) or a double colon (::). A single colon indicates that the label is defined only for the current module (an internal symbol). A double colon indicates that the label is globally defined; that is, the label can be referenced by other object modules (see Section 3.3.2).

Once a label is defined, it cannot be redefined during the source program. If a label is defined more than once, VAX-11 MACRO displays an error message when the label is defined and again when it is referenced.

If a label extends past column 7, it should be placed on a line by itself so that the operator field can start in column 9.

For example:

```
EVAL:   CLRL    R0              ; ROUTINE EVALUATES EXPRESSIONS
ERROR_IN_ARG:                   ; THE ARG-LIST CONTAINS AN ERROR
        INCL    R0              ; INCREMENT ERROR COUNT
TEST::  MOVO    EXP,R1          ; THIS TESTS ROUTINE
                                ; REFERENCED EXTERNALLY
TEST1:  BRW     EXIT_ROU        ; GO TO EXIT ROUTINE
EXP:    .BLKL   50              ; TABLE STORES EXPECTED VALUES
DATA::  .BLKW   25              ; DATA TABLE ACCESSED BY STORE
                                ; ROUTINE IN ALGO MODULE
```

The label field is also used for the symbol in a direct assignment statement (see Section 3.8).

2.2 OPERATOR FIELD

The operator field specifies the action to be performed by the statement. This field can contain either an instruction, or an assembler directive, or a macro call.

When the operator is an instruction, VAX-11 MACRO generates the binary code for that instruction in the object module. The binary codes are listed in Appendix C of this manual; the instruction set is described in the VAX-11 Architecture Handbook. When the operator is a directive, VAX-11 MACRO performs certain control actions or processing operations during source program assembly; the assembler directives are described in Chapters 5 and 6 of this manual. When the operator is a macro call, VAX-11 MACRO expands the macro; macro calls are described in Chapter 6.

Either a space or a tab character may terminate the operator field; however, the tab is the recommended terminating character.

2.3 OPERAND FIELD

The operand field can contain operands for instructions or arguments for assembler directives or macro calls.

Operands for instructions identify the memory locations or the registers that are used by the machine operation. These operands specify the addressing mode for the instruction, as described in Chapter 4 of this manual. The operand field for a specific instruction must contain the number of operands required by that instruction. See the VAX-11 Architecture Handbook for descriptions of the instructions and their operands.

Arguments for a directive must meet the format requirements of the directive. Chapters 5 and 6 describe the directives and the format of their arguments.

Operands for a macro must meet the requirements specified in the macro definition. See the description of the .MACRO directive in Chapter 6.

If two or more operands are specified, they should be separated by commas. VAX-11 MACRO also allows a space or tab to be used as a separator for arguments to any directive that does not accept expressions (see Section 3.5). However, a comma is required to separate operands for instructions and for directives that accept expressions as arguments.

The semicolon that starts the comment field terminates the operand field. If a line does not have a comment field, the operand field is terminated by the end of the line.

2.4 COMMENT FIELD

The comment field contains text that explains the function of the statement. Every line of code should have a comment. Comments do not affect assembly processing or program execution except for messages displayed during assembly by the .ERROR, .PRINT, and .WARN directives (see descriptions in Chapter 5).

The comment field must be preceded by a semicolon and is terminated by the end of the line. The comment field can contain any printable ASCII character (see Appendix A).

If a comment does not fit on one line, it can be continued on the next, but the continuation must be preceded by another semicolon. A comment can appear on a line by itself.

The text of a comment normally conveys the meaning rather than the action of the statement. The instruction MOVAL BUF_PTR_1,R7, for instance, should have a comment such as "GET POINTER TO FIRST BUFFER" not "MOVE ADDRESS OF BUF_PTR_1 TO R7."

For example:

```
        MOVAL   STRING_DES_1,R0         ; GET ADDRESS OF STRING
                                        ; DESCRIPTOR
        MOVZWL  (R0),R1                 ; GET LENGTH OF STRING
        MOVL    4(R0),R0                ; GET ADDRESS OF STRING
```

CHAPTER 3

THE COMPONENTS OF MACRO SOURCE STATEMENTS

This chapter describes the components of VAX-11 MACRO source statements. These components consist of the character set; numbers; symbols; local labels; terms and expressions; unary and binary operators; direct assignment statements; and the current location counter.

3.1 CHARACTER SET

The following characters can be used in VAX-11 MACRO source statements:

- Both uppercase and lowercase letters (A through Z, a through z) are accepted. However, the assembler considers lowercase letters equivalent to uppercase except when they appear in ASCII strings.
- The digits 0 through 9.
- The special characters listed in Table 3-1.

Table 3-1
Special Characters Used in VAX-11 MACRO Statements

Character	Character Name	Function
_	Underline	Character in symbol names
$	Dollar sign	Character in symbol names
.	Period	Character in symbol names, current location counter, and decimal point
:	Colon	Label terminator
=	Equal sign	Direct assignment operator and macro keyword argument terminator
	Tab	Field terminator
	Space	Field terminator

(continued on next line)

Table 3-1 (Cont.)
Special Characters Used in VAX-11 MACRO Statements

Character	Character Name	Function
#	Number sign	Immediate addressing mode indicator
@	At sign	Deferred addressing mode indicator and arithmetic shift operator
,	Comma	Field, operand, and item separator
;	Semicolon	Comment field indicator
+	Plus sign	Autoincrement addressing mode indicator, unary plus operator, and arithmetic addition operator
-	Minus sign or hyphen	Autodecrement addressing mode indicator, unary minus operator, arithmetic subtraction operator, and line continuation indicator
*	Asterisk	Arithmetic multiplication operator
/	Slash	Arithmetic division operator
&	Ampersand	Logical AND operator
!	Exclamation point	Logical inclusive OR operator
\	Backslash	Logical exclusive OR and numeric conversion indicator in macro arguments
^	Circumflex	Unary operators and macro argument delimiter
[]	Square brackets	Index addressing mode and repeat count indicators
()	Parentheses	Register deferred addressing mode indicators
<>	Angle brackets	Argument or expression grouping delimiters
?	Question mark	Created label indicator in macro arguments
'	Apostrophe	Macro argument concatenation indicator
%	Percent sign	Macro string operators

Table 3-2 defines the separating characters used in VAX-11 MACRO.

Table 3-2
Separating Characters in VAX-11 MACRO Statements

Character	Character Name	Usage
	Space or tab	Separator between statement fields. Spaces within expressions (see Section 3.5) are ignored.
	Comma	Separator between symbolic arguments within the operand field. Multiple expressions in the operand field must be separated by commas.

3.2 NUMBERS

Numbers can be integers, floating-point numbers, or packed decimal strings.

3.2.1 Integers

Integers can be used in any expression including expressions in operands and in direct assignment statements (Section 3.5 describes expressions).

Format

snn

s

An optional sign: plus sign (+) for positive numbers (the default) or minus sign (-) for negative numbers.

nn

A string of numeric characters that are legal for the current radix.

VAX-11 MACRO interprets all integers in the source program as decimal unless the number is preceded by a radix control operator (see Section 3.6.1).

Integers must be in the range of -2147483648 through 2147483647 for signed data or in the range of 0 through 4294967295 for unsigned data.

Negative numbers must be preceded by a minus sign; VAX-11 MACRO translates such numbers into 2's complement form. In positive numbers, the plus sign is optional.

3.2.2 Floating-Point Numbers

A floating-point number can be used in the .F_FLOATING (.FLOAT), .D_FLOATING, (.DOUBLE) .G_FLOATING, and .H_FLOATING directives (described in Chapter 5) or as an operand in a floating-point instruction. A floating-point number cannot be used in an expression or with a unary or binary operator except the unary plus, unary minus, and unary floating-point operator (^F). Sections 3.6 and 3.7 describe unary and binary operators.

A floating-point number can be specified with or without an exponent.

Formats

Floating-point number without exponent:

```
snn
snn.nn
snn.
```

Floating-point number with exponent:

```
snnEsnn
snn.nnEsnn
snn.Esnn
```

s

An optional sign.

nn

A string of decimal digits in the range of 0 through 9.

The decimal point can appear anywhere to the right of the first digit. However, note that a floating-point number cannot start with a decimal point because VAX-11 MACRO will treat the number as a user-defined symbol (see Section 3.3.2).

Floating-point numbers can be single-precision (32-bit), double-precision (64-bit), or extended-precision (128-bit) quantities. The degree of precision is 7 digits for single-precision numbers, 16 digits for double-precision numbers, and 33 digits for extended precision numbers.

The magnitude of a nonzero floating-point number cannot be smaller than approximately 0.29E-38 or greater than approximately 1.7E38.

Single-precision floating-point numbers can be rounded (by default) or truncated. The .ENABLE and .DISABLE directives (described in Chapter 5) control whether single-precision floating-point numbers are rounded or truncated. Double-precision and extended-precision floating point numbers are always rounded.

The VAX-11 Architecture Handbook describes the internal format of floating-point numbers.

3.2.3 Packed Decimal Strings

A packed decimal string can be used only in the .PACKED directive (described in Chapter 5).

Format

snn

s
: An optional sign.

nn
: A string of from 1 to 31 decimal digits in the range of 0 through 9.

A packed decimal string cannot have a decimal point or an exponent.

The <u>VAX-11 Architecture Handbook</u> describes the internal format of packed decimal strings.

3.3 SYMBOLS

Three types of symbols can be used in VAX-11 MACRO source programs: permanent symbols, user-defined symbols, and macro names.

3.3.1 Permanent Symbols

Permanent symbols consist of instruction mnemonics (see Appendix C), VAX-11 MACRO directives (see Chapters 5 and 6), and register names. Instruction mnemonics and directives need not be defined before being used in the operator field of a VAX-11 MACRO source statement. Register names need not be defined before being used in the addressing modes (see Chapter 4). Register names cannot be redefined; that is, no user-defined symbol can have one of the register names listed below.

The 16 general registers of the VAX-11 processor can be expressed in a source program only as follows:

Register Name	Processor Register
R0	General register 0
R1	General register 1
R2	General register 2
.	.
.	.
.	.
R11	General register 11
R12 or AP	General register 12 or argument pointer. If R12 is used as an argument pointer, the name AP is recommended; if R12 is used as a general register, the name R12 is recommended.

Register Name	Processor Register
FP	Frame pointer
SP	Stack pointer
PC	Program counter

3.3.2 User-defined Symbols and Macro Names

User-defined symbols can be used as labels or can be equated to a specific value by a direct assignment statement (see Section 3.8).

User-defined symbols also can be used in any expression (see Section 3.5).

The following rules govern the creation of user-defined symbols:

- User-defined symbols can be composed of alphanumeric characters, underlines (_), dollar signs ($), and periods (.). Any other character terminates the symbol.
- The first character of a symbol must not be a number.
- The symbol must be no more than 31 characters long and must be unique.

In addition, by DIGITAL convention:

- The dollar sign ($) is reserved for names defined by DIGITAL. This convention ensures that a user-defined name (which does not have a dollar sign) will not conflict with a DIGITAL-defined name (which does have a dollar sign).
- The period (.) should not be used in any global symbol name (see Section 3.3.3) because other languages, such as FORTRAN, do not allow periods in symbol names.

Macro names follow the same rules and conventions as user-defined symbols (see the description of the .MACRO directive in Chapter 6 for more information on macro names). User-defined symbols and macro names do not conflict; that is, the same name can be used for a user-defined symbol and a macro. However, to avoid confusion, user-defined symbols and macros should be given different names.

3.3.3 Determining Symbol Values

The value of a symbol depends on its use in the program. VAX-11 MACRO uses a different method to determine the values of symbols in the operator field than it uses to determine the values of symbols in the operand field.

A symbol in the operator field can be either a permanent symbol or a macro name. VAX-11 MACRO searches for a symbol definition in the following order:

- Previously defined macro names
- User-defined opcode (see the .OPDEF description in Chapter 5)

- Permanent symbols (instructions and directives)
- Macro libraries

This search order allows permanent symbols to be redefined as macro names. If a symbol in the operator field is not defined as a macro or a permanent symbol, the assembler displays an error message.

A symbol in the operand field must be either a user-defined symbol or a register name.

User-defined symbols can be either local (internal) symbols or global (external) symbols. Whether symbols are local or global depends on their use in the source program.

A local symbol can be referenced only in the module in which it is defined. If local symbols with the same names are defined in different modules, the symbols are completely independent. A global symbol's definition, however, can be referenced from any module in the program.

Normally, VAX-11 MACRO treats all user-defined symbols as local when they are defined. However, a symbol definition can be explicitly declared to be global by any one of the following three methods:

- Use of the double colon (::) in defining a label (see Section 2.1)
- Use of the double equal sign (==) in a direct assignment statement (see Section 3.8)
- Use of the .GLOBAL, .ENTRY, or .WEAK directive (see Chapter 5)

When a symbol is referenced within the module in which it is defined, VAX-11 MACRO considers the reference an internal reference. When a symbol is referenced within a module in which it is not defined, VAX-11 MACRO considers the reference an external reference (that is, the symbol is defined in another module). The .DISABLE directive can be used to make references to symbols not defined in the current module illegal. In this case, the .EXTERNAL directive must be used to specify that the reference is an external reference. See Chapter 5 for descriptions of the .DISABLE and .EXTERNAL directives.

3.4 LOCAL LABELS

Local labels are used to identify addresses within a block of source code.

Format

```
nn$
```

nn
: A decimal integer in the range of 1 through 65535.

Local labels can be used in the same way as user-defined symbol labels, but with the following differences:

- Local labels cannot be referenced outside the block of source code in which they appear.
- Local labels can be reused in another block of source code.

- Local labels do not appear in the symbol tables and, thus, cannot be accessed by the debugger.

- Local labels cannot be used in .END (see Chapter 5).

By convention, local labels are positioned like statement labels: left-justified in the source text. Although local labels can appear in the program in any order, by convention, the local labels in any block of source code should be in numeric order.

Local labels are useful as branch addresses when the address is used only within the block. Local labels can be used to distinguish between addresses that are referenced only in a small block of code and addresses that are referenced elsewhere in the module. A disadvantage of local labels is that their numeric names cannot provide any indication of their purpose. Consequently, local labels should not be used to label sequences of statements that are logically unrelated; user-defined symbols should be used instead.

DIGITAL recommends that users create local labels only in the range of 1$ to 29999$ because the assembler automatically creates local labels in the range of 30000$ to 65535$ for use in macros (see Section 6.1.6).

The local label block in which a local label is valid is delimited by the following statements:

- A user-defined label

- A .PSECT directive (see Chapter 5)

- The .ENABLE and .DISABLE directives (see Chapter 5) which can extend a local label block beyond user-defined labels and .PSECT directives

A local label block is usually delimited by two user-defined labels. However, the .ENABLE LOCAL_BLOCK directive starts a local block that is terminated only by one of the following:

- A second .ENABLE LOCAL_BLOCK directive

- A .DISABLE LOCAL_BLOCK directive followed by a user-defined label or a .PSECT directive

Although local label blocks can extend from one program section to another, DIGITAL recommends that local labels in one program section not be referenced from another program section. User-defined symbols should be used instead.

Local labels can be preserved for future reference with the context of the program section in which they are defined; see the descriptions of the .SAVE_PSECT [LOCAL_BLOCK] directive and the .RESTORE_PSECT directive in Chapter 5.

An example showing the use of local labels follows.

```
RPSUB:  MOVL    AMOUNT,R0           ; STARTS LOCAL LABEL BLOCK
10$:    SUBL2   DELTA,R0            ; DEFINE LOCAL LABEL 10$
        BGTR    10$                 ; CONDITIONAL BRANCH TO LOCAL LABEL
        ADDL2   DELTA,R0            ; EXECUTED WHEN R0 NOT > 0
COMP:   MOVL    MAX,R1              ; ENDS PREVIOUS LOCAL LABEL
        CLRL    R2                  ; BLOCK AND STARTS NEW ONE
10$:    CMPL    R0,R1               ; DEFINE NEW LOCAL LABEL 10$
        BGTR    20$                 ; CONDITIONAL BRANCH TO LOCAL LABEL
        SUBL    INCR,R0             ; EXECUTED WHEN R0 NOT > R1
        INCL    R2                  ; .  .  .
        BRB     10$                 ; UNCONDITIONAL BRANCH TO LOCAL LABEL
20$:    MOVL    R2,COUNT            ; DEFINE LOCAL LABEL
        BRW     TEST                ; UNCONDITIONAL BRANCH TO
                                    ; USER-DEFINED LABEL
        .ENABLE LOCAL_BLOCK         ; START LOCAL LABEL BLOCK
ENTR1:  POPR    #^M<R0,R1,R2>       ; THAT WILL NOT BE TERMINATED
        ADDL3   R0,R1,R3            ; BY A USER-DEFINED LABEL
        BRB     10$                 ; BRANCH TO LOCAL LABEL THAT IS AFTER
                                    ; A USER-DEFINED LABEL
ENTR2:  SUBL2   R2,R3               ; DOES NOT START A NEW
                                    ; LOCAL LABEL BLOCK
10$:    SUBL2   R2,R3               ; DEFINE LOCAL LABEL
        BGTR    20$                 ; CONDITIONAL BRANCH TO LOCAL LABEL
        INCL    R0                  ; EXECUTED WHEN R2 NOT > R3
        BRB     NEXT                ; UNCONDITIONAL BRANCH TO
                                    ; USER-DEFINED LABEL
20$:    DECL    R0                  ; DEFINE LOCAL LABEL
        .DISABLE LOCAL_BLOCK        ; DIRECTIVE FOLLOWED
NEXT:   CLRL    R4                  ; BY USER-DEFINED LABEL TERMINATES
                                    ; LOCAL LABEL BLOCK
```

3.5 TERMS AND EXPRESSIONS

A term can be any one of the following:

- A number
- A symbol
- The current location counter (see Section 3.9)
- A textual operator followed by text (see Section 3.6.2)
- Any of the above preceded by a unary operator (see Section 3.6)

VAX-11 MACRO evaluates terms as longword (4-byte) values. If an undefined symbol is used as a term, the linker determines the term's value. The current location counter (.) has the value of the location counter at the start of the current operand.

Expressions are combinations of terms joined by binary operators (see Section 3.7) and evaluated as longword (4-byte) values. VAX-11 MACRO evaluates expressions from left to right with no operator precedence rules. However, angle brackets (<>) can be used to change the order of evaluation. Any part of an expression that is enclosed in angle brackets is first evaluated to a single value, which is then used in evaluating the complete expression. For example, the expressions A*B+C and A*<B+C> are different. Angle brackets can also be used to apply a unary operator to an entire expression, such as -<A+B>.

Note that unary operators are considered part of a term; thus, VAX-11 MACRO performs the action indicated by a unary operator before it performs the action indicated by any binary operator.

Expressions fall into three categories: relocatable, absolute, and external (global).

- An expression is relocatable if its value is fixed relative to the start of the program section in which it appears. The current location counter is relocatable in a relocatable program section.
- An expression is absolute if its value is an assembly-time constant. An expression whose terms are all numbers is absolute. An expression that consists of a relocatable term minus another relocatable term from the same program section is absolute, because such an expression reduces to an assembly-time constant.
- An expression is external if it contains one or more symbols that are not defined in the current module.

Any type of expression can be used in most macro statements, but restrictions are placed on expressions used in:

- .ALIGN alignment directive
- .BLKx storage allocation directives
- .IF and .IIF conditional assembly block directives
- .REPEAT repeat block directive
- .OPDEF opcode definition directive
- .ENTRY entry point directive
- .BYTE, .LONG, .WORD, .SIGNED_BYTE, and .SIGNED_WORD directive repetition factors
- Direct assignment statements (see Section 3.8)

See Chapter 5 for descriptions of the directives listed above, except .REPEAT which is described in Chapter 6. Expressions used in these directives and in direct assignment statements can only contain symbols that have been previously defined in the current module. They cannot contain either external symbols or symbols defined later in the current module. In addition, the expressions in these directives must be absolute. Expressions in direct assignment statements can be relocatable.

An example showing the use of expressions follows.

```
A = 2*100                               ; 2*100 IS AN ABSOLUTE EXPRESSION
        .BLKB   A+50                    ; A+50 IS AN ABSOLUTE EXPRESSION AND
                                        ; CONTAINS NO UNDEFINED SYMBOLS
LAB:    .BLKW   A                       ; LAB IS RELOCATABLE
HALF = LAB+<A/2>                        ; LAB+<A/2> IS A RELOCATABLE
                                        ; EXPRESSION AND CONTAINS NO
                                        ; UNDEFINED SYMBOLS
LAB2:   .BLKB   LAB2-LAB                ; LAB2-LAB IS AN ABSOLUTE EXPRESSION
                                        ; AND CONTAINS NO UNDEFINED SYMBOLS
        .WORD   LAB3-LAB2               ; LAB3-LAB2 IS AN ABSOLUTE EXPRESSIO
                                        ; BUT CONTAINS THE SYMBOL LAB3
                                        ; THAT IS DEFINED LATER IN THIS MODU
LAB3:   .WORD   TST+LAB+2               ; TST+LAB+2 IS AN EXTERNAL EXPRESSIO
                                        ; BECAUSE TST IS AN EXTERNAL SYMBOL
```

3.6 UNARY OPERATORS

A unary operator modifies a term or an expression, and indicates an action to be performed on that term or expression. Expressions modified by unary operators must be enclosed in angle brackets. Unary operators can be used to indicate whether a term or expression is positive or negative (if unary plus or minus is not specified, the value is assumed to be plus, by default). In addition, unary operators perform radix conversion, textual conversion (including ASCII conversion), and numeric control operations, as described in Sections 3.6.1 through 3.6.3. Table 3-3 summarizes the unary operators.

Table 3-3
Unary Operators

Unary Operator	Operator Name	Example	Operation
+	Plus sign	+A	Results in the positive value of A
-	Minus sign	-A	Results in the negative (2's complement) value of A
^B	Binary	^B11000111	Specifies that 11000111 is a binary number
^D	Decimal	^D127	Specifies that 127 is a decimal number
^O	Octal	^O34	Specifies that 34 is an octal number
^X	Hexadecimal	^XFCF9	Specifies that FCF9 is a hexadecimal number
^A	ASCII	^A/ABC/	Produces an ASCII string; the characters between the matching delimiters are converted to ASCII representation
^M	Register mask	#^M<R3,R4,R5>	Specifies the registers R3, R4, and R5 in the register mask
^F	Floating point	^F3.0	Specifies that 3.0 is a floating-point number
^C	Complement	^C24	Produces the 1's complement value of 24 (decimal)

More than one unary operator can be applied to a single term or to an expression enclosed in angle brackets. For example:

```
-+-A
```

This construct is equivalent to:

```
-<+<-A>>
```

3.6.1 Radix Control Operators

VAX-11 MACRO accepts terms or expressions in four different radixes: binary, decimal, octal, and hexadecimal. The default radix is decimal. Expressions modified by radix control operators must be enclosed in angle brackets.

Formats

```
^Bnn
^Dnn
^Onn
^Xnn
```

nn

A string of characters that are legal in the specified radix. The legal characters for each radix are listed below.

Format	Radix Name	Legal Characters
^Bnn	Binary	0 and 1
^Dnn	Decimal	0 through 9
^Onn	Octal	0 through 7
^Xnn	Hexadecimal	0 through 9 and A through F

Radix control operators can be included in the source program anywhere a numeric value is legal. A radix control operator affects only the term or expression immediately following it, causing that term or expression to be evaluated in the specified radix.

For example:

```
.WORD   ^B00001101              ; BINARY RADIX
.WORD   ^D123                   ; DECIMAL RADIX (DEFAULT)
.WORD   ^O47                    ; OCTAL RADIX
.WORD   <A+^O13>                ; 13 IS IN OCTAL RADIX
.LONG   ^X<F1C3+FFFFF-20>       ; ALL NUMBERS IN EXPRESSION
                                ; ARE IN HEXADECIMAL RADIX
```

The circumflex cannot be separated from the B, D, O, or X that follows it, but the entire radix control operator can be separated by spaces and tabs from the term or expression that is to be evaluated in that radix.

The decimal operator, the default, is needed only within an expression that has another radix control operator. In the following example, the 16 would be interpreted as an octal number if the ^D operator did not precede it:

```
.LONG   ^O<10000 + 100 + ^D16>
```

3.6.2 Textual Operators

The textual operators are the ASCII operator (^A) and the register mask operator (^M).

3.6.2.1 ASCII Operator

- The ASCII operator converts a string of printable characters to their 8-bit ASCII values and stores them one character to a byte. The string of characters must be enclosed in a pair of matching delimiters.

The delimiters can be any printable character except the space, tab, or semicolon (;). Although alphanumeric characters can be used as delimiters, nonalphanumeric characters should be used to avoid confusion.

Format

```
^Astring
```

string
    A delimited ASCII string from 1 through 16 characters long.

The delimited ASCII string must not be larger than the data type of the operand. For example, if the ^A operator occurs in an operand in a MOVW instruction (the data type is a word), the delimited string cannot be more than two characters.

For example:

```
        MOVL    #^A/ABCD/,R0            ; MOVES CHARACTERS A,B,C,D
                                        ; INTO R0 RIGHT JUSTIFIED WITH
                                        ; "A" IN LOW-ORDER BYTE AND "D"
                                        ; IN HIGH-ORDER BYTE
        CMPW    #^A/XY/,R0              ; COMPARES X AND Y AS ASCII
                                        ; CHARACTERS WITH CONTENTS OF LOW
                                        ; ORDER 2 BYTES OF R0
        .QUAD   ^A%1234/678%            ; GENERATES 8 BYTES OF ASCII DATA
        MOVL    #^A/AB/,R0              ; MOVE ASCII CHARACTERS AB INTO
                                        ; R0; "A" IN LOW-ORDER BYTE; "B"
                                        ; IN NEXT; AND ZERO THE 2 HIGH-
                                        ; ORDER BYTES
```

3.6.2.2 Register Mask Operator

- The register mask operator converts a register name or a list of register names enclosed in angle brackets into a 1- or 2-byte register mask. The register mask is used by the PUSHR and POPR instructions and the .ENTRY and .MASK directives (see Chapter 5).

Formats

```
^Mreg-name
^M<reg-name-list>
```

reg-name
    One of the register names or the DV or IV arithmetic trap enable specifiers.

reg-name-list
    A list of register names and/or the DV and IV arithmetic trap enable specifiers, separated by commas.

The register mask operator sets a bit in the register mask for every register name or arithmetic trap enable specified in the list. The bits corresponding to each register name and arithmetic trap enable specifier are listed below.

Register Name	**Arithmetic Trap Enable**	**Bits**
R0 through R11		0 through 11 (respectively)
R12 or AP		12
FP		13
SP	IV	14
	DV	15

When the register mask operator is used in a POPR or PUSHR instruction, R0 through R11, R12 or AP, FP, and SP can be specified. The PC register name and the IV and DV arithmetic trap enable specifiers cannot be specified.

When the register mask operator is used in the .ENTRY or .MASK directives, R2 through R11 and the IV and DV arithmetic trap enable specifiers can be specified. However, R0, R1, FP, SP, and PC cannot be specified. IV sets the integer overflow trap, and DV sets the decimal string overflow trap.

See the VAX-11 Architecture Handbook for more information on register masks and arithmetic trap enable specifiers.

For example:

```
        .ENTRY  RT1,^M<R3,R4,R5,R6,IV>  ; SAVE REGISTERS R3,R4
                                        ; R5, AND R6 AND SET THE
                                        ; INTEGER OVERFLOW TRAP
        PUSHR   #^M<R0,R1,R2,R3>        ; SAVE REGISTERS R0,R1,
                                        ; R2, AND R3
        POPR    #^M<R0,R1,R2,R3>        ; RESTORE R0,R1,R2, AND R3
```

3.6.3 Numeric Control Operators

The numeric control operators are the floating-point operator (^F) and the complement operator (^C).

3.6.3.1 Floating-Point Operator

3.6.3.1 **Floating-Point Operator** - The floating-point operator accepts a floating-point number and converts it to its internal representation (a 4-byte value). This value can be used in any expression. VAX-11 MACRO does not perform floating-point expression evaluation.

Format

```
^Fliteral
```

literal
    A floating-point number (see Section 3.2.2).

The floating-point operator is useful because it allows a floating-point number in an instruction that accepts integers.

For example:

```
        MOVL    #^F3.7,R0            ; NOTE THE RECOMMENDED INSTRUCTION
                                     ; TO MOVE THIS FLOATING-POINT NUMBER
        MOVF    #3.7,R0              ; IS THE MOVF INSTRUCTION
```

3.6.3.2 **Complement Operator** - The complement operator produces the 1's complement of the specified value.

Format

^Cterm

term
: Any term or expression. If an expression is specified, it must be enclosed in angle brackets.

VAX-11 MACRO evaluates the term or expression as a 4-byte value before complementing it.

For example:

```
        .LONG   ^C^XFF               ; PRODUCES FFFFFF00 (HEX)
        .LONG   ^C25                 ; PRODUCES COMPLEMENT OF
                                     ; 25 (DEC) WHICH IS
                                     ; FFFFFFE6 (HEX)
```

3.7 BINARY OPERATORS

In contrast to unary operators, binary operators specify actions to be performed on two terms or expressions. Expressions must be enclosed in angle brackets. Table 3-4 summarizes the binary operators.

Table 3-4
Binary Operators

Binary Operator	Operator Name	Example	Operation
+	Plus sign	A+B	Addition
-	Minus sign	A-B	Subtraction
*	Asterisk	A*B	Multiplication
/	Slash	A/B	Division
@	At sign	A@B	Arithmetic shift
&	Ampersand	A&B	Logical AND
!	Exclamation point	A!B	Logical inclusive OR
\	Backslash	A\B	Logical exclusive OR

All binary operators have equal priority. Terms or expressions can be grouped for evaluation by enclosing them in angle brackets. The enclosed terms and expressions are then evaluated first, and remaining operations are performed from left to right. For example:

```
        .LONG    1+2*3                    ; EQUALS 9
        .LONG    1+<2*3>                  ; EQUALS 7
```

Note that a 4-byte result is returned from all binary operations. If a 1-byte or 2-byte operand is used, the result is the low-order byte(s) of the 4-byte result. VAX-11 MACRO displays an error message if the truncation causes a loss of significance.

The following sections describe the arithmetic shift, logical AND, logical inclusive OR, and logical exclusive OR operators in more detail.

3.7.1 Arithmetic Shift Operator

The arithmetic shift operator (@) is used to perform left and right arithmetic shift of arithmetic quantities. The first argument is shifted left or right the number of bit positions specified by the second argument. If the second argument is positive, the first argument is shifted left; if the second argument is negative, the first argument is shifted right. When the first argument is shifted left, the low-order bits are set to 0; and when the first argument is shifted right, the high-order bits are set to the value of the original high-order bit (the sign bit).

For example:

```
        .LONG    ^B101@4                  ; YIELDS 1010000 (BINARY)
        .LONG    1@2                      ; YIELDS 100 (BINARY)
        MOVL     #<^B1100000@-5>,R0       ; YIELDS 11 (BINARY)

A = 4

        .LONG    1@A                      ; YIELDS 10000 (BINARY)
        .LONG    ^X1234@-A                ; YIELDS 123(HEX)
```

3.7.2 Logical AND Operator

The logical AND operator (&) takes the logical AND of two operands.

For example:

```
A = ^B1010
B = ^B1100
        .LONG    A&B                      ; YIELDS 1000 (BINARY)
```

3.7.3 Logical Inclusive OR Operator

The logical inclusive OR operator (!) takes the logical inclusive OR of two operands.

For example:

```
A = ^B1010
B = ^B1100
        .LONG   A!B                     ; YIELDS 1110 (BINARY)
```

3.7.4 Logical Exclusive OR Operator

The logical exclusive OR operator (\) takes the logical exclusive OR of two arguments.

For example:

```
A = ^B1010
B = ^B1100
        .LONG   A\B                     ; YIELDS 0110 (BINARY)
```

3.8 DIRECT ASSIGNMENT STATEMENTS

A direct assignment statement equates a symbol to a specific value. Unlike a symbol that is used as a label, a symbol defined with a direct assignment statement can be redefined as many times as desired.

Formats

```
symbol=expression
symbol==expression
```

symbol
: A user-defined symbol.

expression
: An expression that does not contain any undefined symbols (see Section 3.5).

The format with a single equal sign (=) defines a local symbol and the format with a double equal sign (==) defines a global symbol. See Section 3.3.3 for more information about local and global symbols.

The following three syntactic rules apply to direct assignment statements:

- An equal sign (=) or double equal sign (==) must separate the symbol from the expression defining the symbol's value. Spaces preceding and/or following the direct assignment operators have no significance in the resulting value.
- Only one symbol can be defined in a single direct assignment statement.
- A direct assignment statement can be followed only by a comment field.

In addition, by DIGITAL convention, the symbol in a direct assignment statement is placed in the label field.

For example:

```
A = = 1                         ; THE SYMBOL 'A' IS GLOBALLY
                                ; EQUATED TO THE VALUE 1
B = A@5                         ; THE SYMBOL 'B' IS EQUATED
                                ; TO 1@5 OR 20(HEX)
C = 127*10                      ; THE SYMBOL 'C' IS EQUATED
                                ; TO 1270(DEC)
D = ^X100/^X10                  ; THE SYMBOL 'D' IS EQUATED
                                ; TO 10(HEX)
```

3.9 CURRENT LOCATION COUNTER

The period (.), the symbol for the current location counter, always has the value of the address of the current byte. VAX-11 MACRO sets the current location counter to 0 at the beginning of the assembly and at the beginning of each new program section.

Every VAX-11 MACRO source statement that allocates memory in the object module increments the value of the current location counter by the number of bytes allocated. For example, the directive .LONG 0 increments the current location counter by 4. However, a direct assignment statement, except the special form described below, does not increase the current location counter because no memory is allocated.

The current location counter can be explicitly set by a special form of the direct assignment statement. The location counter can be either incremented or decremented. Explicitly setting the location counter is often useful when defining data structures. Data storage areas should not be reserved by explicitly setting the location counter; the .BLKx directives should be used instead (see Chapter 5).

Format

```
.=expression
```

expression
: An expression that does not contain any undefined symbols (see Section 3.5).

In a relocatable program section, the expression must be relocatable; that is, the expression must be relative to an address in the current program section. It can be relative to the current location counter.

For example:

```
.  = .+40                       ; MOVES LOCATION COUNTER
                                ; FORWARD
```

When a program section previously defined in the current module is continued, the current location counter is set to the last value of the current location counter in that program section.

When the current location counter is used in the operand field of an instruction, the current location counter has the value of the address of that operand--it does not have the value of the address of the beginning of the instruction. For this reason, the current location counter is not normally used as a part of the operand specifier.

CHAPTER 4

ADDRESSING MODES

This chapter summarizes the VAX-11 addressing modes and contains examples of VAX-11 MACRO statements that use these addressing modes. The VAX-11 Architecture Handbook describes the addressing modes in detail.

There are four types of addressing modes:

- General Register
- Program Counter
- Index
- Branch

Although index mode is a general register mode, it is considered a separate type of mode because it can be used only in combination with another type of mode.

Table 4-1 summarizes the addressing modes.

4.1 GENERAL REGISTER MODES

The general register modes use registers R0 through R12, AP (the same as R12), FP, and SP.

There are eight general register modes:

- Register
- Register Deferred
- Autoincrement
- Autoincrement Deferred
- Autodecrement
- Displacement
- Displacement Deferred
- Literal

Table 4-1
Addressing Modes

Type	Addressing Mode	Format*	Hexa-decimal Value	Description	Indexable?
General Register	Register	Rn	5	Register contains the operand	No
	Register Deferred	(Rn)	6	Register contains the address of the operand	Yes
	Autoincrement	(Rn)+	8	Register contains the address of the operand; the processor increments the register contents by the size of the operand data type	Yes
	Autoincrement Deferred	@(Rn)+	9	Register contains the address of the operand address; the processor increments the register contents by 4	Yes
	Autodecrement	-(Rn)	7	The processor decrements the register contents by the size of the operand data type; the register then contains the address of the operand	Yes
	Displacement	dis(Rn) B^dis(Rn) W^dis(Rn) L^dis(Rn)	 A C E	The sum of the contents of the register and the displacement is the address of the operand; B^, W^,and L^ indicate byte, word, and longword displacement, respectively	Yes
	Displacement Deferred	@dis(Rn) @B^dis(Rn) @W^dis(Rn) @L^dis(Rn)	 B D F	The sum of the contents of the register and the displacement is the address of the operand address; B^, W^, and L^ indicate byte, word, and longword displacement, respectively	Yes
General Register (Cont.)	Literal	#literal S^#literal	 0-3	The literal specified is the the operand; the literal is stored as a short literal	No
Program Counter	Relative	address B^address W^address L^address	 A C E	The address specified is the address of the operand; the address specified is stored as a displacement from PC; B^, W^, and L^ indicate byte, word, and longword displacement, respectively	Yes
	Relative Deferred	@address @B^address @W^address @L^address	 B D F	The address specified is the address of the operand address; the address specified is stored as a displacement from PC; B^, W^, and L^ indicate byte, word, and longword displacement, respectively	Yes
	Absolute	@#address	9	The address specified is the address of the operand; the address specified is stored as an absolute virtual address (not as a displacement)	Yes
	Immediate	#literal I^#literal	 8	The literal specified is the operand; the literal is stored as a byte, word, longword, or quadword	No
	General	G^address	-	The address specified is the address of the operand; if the address is defined as relocatable, the linker stores the address as a displacement from PC; if the address is defined as an absolute virtual address, the linker stores the address as an absolute value	Yes

(continued on next page)

Table 4-1 (Cont.)
Addressing Modes

Type	Addressing Mode	Format*	Hexa-decimal Value	Description	Indexable?
Index	Index	base-mode [Rx]	4	The base-mode specifies the base address and the register specifies the index; the sum of the base address and the product of the contents of Rx and the size of the operand data type is the address of the operand; base-mode can be any addressing mode except register, immediate, literal, index, or branch	No
Branch	Branch	address	-	The address specified is the operand; this address is stored as a displacement to PC; branch mode can only be used with the branch instructions	No

* Key:

Rn

Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rn.

Rx

Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rx. Rx cannot be the same as the Rn specified in the base-mode for certain base modes (see Section 4.3).

dis

An expression specifying a displacement.

address

An expression specifying an address.

literal

An expression, an integer constant, or a floating-point constant.

4.1.1 Register Mode

In register mode, the operand is the contents of the specified register, except in the following cases:

- For quadword, D_floating, G_floating or variable-bit field operands, the operand is the contents of register n concatenated with the contents of register n+1.
- For octaword and H_floating operands, the operand is the contents of register n concatenated with the contents of registers n+1, n+2, and n+3.

In each of these cases, the least significant bytes of the operand are in register n and the most significant bytes are in the highest register used, either n+1 or n+3.

The results of the operation are unpredictable if PC is used in register mode or if the use of a large data type extends the operand into the PC.

Formats

```
Rn
AP
FP
SP
```

n
: A number in the range of 0 through 12.

Example

```
        CLRB    R0              ; CLEAR LOWEST BYTE OF R0
        CLRQ    R1              ; CLEAR R1 AND R2
        TSTW    R10             ; TEST LOWER WORD OF R10
        INCL    R4              ; ADD 1 TO R4
```

4.1.2 Register Deferred Mode

In register deferred mode, the register contains the address of the operand. Register deferred mode can be used with index mode (see Section 4.3).

Formats

```
(Rn)
(AP)
(FP)
(SP)
```

n
: A number in the range of 0 through 12.

Example

```
        MOVAL   LDATA,R3        ; MOVE ADDRESS OF LDATA TO R3
        CMPL    (R3),R0         ; COMPARE VALUE AT LDATA TO R0
        BEQL    10$             ; IF THEY ARE THE SAME, IGNORE
        CLRL    (R3)            ; CLEAR LONGWORD AT LDATA
10$:    MOVL    (SP),R1         ; COPY TOP ITEM OF STACK INTO R1
        MOVZBL  (AP),R4         ; GET NUMBER OF ARGUMENTS IN CALL
```

4.1.3 Autoincrement Mode

In autoincrement mode, the register contains the address of the operand. After evaluating the operand address contained in the register, the processor increments that address by the size of the operand data type. The processor increments the contents of the register by 1, 2, 4, 8, or 16 for a byte, word, longword, quadword, or octaword operand, respectively.

Autoincrement mode can be used with index mode (see Section 4.3), but the index register cannot be the same as the register specified in autoincrement mode.

Formats

```
(Rn)+
(AP)+
(FP)+
(SP)+
```

n

A number in the range of 0 through 12.

Example

```
MOVAL   TABLE,R1          ; GET ADDRESS OF TABLE
CLRQ    (R1)+             ; CLEAR FIRST AND SECOND LONGWORDS
CLRL    (R1)+             ; AND THIRD LONGWORD IN TABLE
                          ; LEAVE R1 POINTING TO TABLE +12
MOVAB   BYTARR,R2         ; GET ADDRESS OF BYTARR
INCB    (R2)+             ; INCREMENT FIRST BYTE OF BYTARR
INCB    (R2)+             ; AND SECOND
XORL3   (R3)+,(R4)+,(R5)+ ; EXCLUSIVE-OR THE TWO LONGWORDS
                          ; WHOSE ADDRESSES ARE STORED IN
                          ; R3 AND R4 AND STORE RESULT IN
                          ; ADDRESS CONTAINED IN R5, THEN
                          ; ADD 4 TO R3, R4, AND R5
```

4.1.4 Autoincrement Deferred Mode

In autoincrement deferred mode, the register contains an address that is the address of the operand address (a pointer to the operand). After evaluating the operand address, the processor increments the contents of the register by 4 (the size in bytes of an address).

Autoincrement deferred mode can be used with index mode (see Section 4.3), but the index register cannot be the same as the register specified in autoincrement deferred mode.

Formats

```
@(Rn)+
@(AP)+
@(FP)+
@(SP)+
```

n

A number in the range of 0 through 12.

Example

```
MOVAL   PNTLIS,R2       ; GET ADDRESS OF POINTER LIST
CLRQ    @(R2)+          ; CLEAR QUADWORD POINTED TO BY
                        ; FIRST ABSOLUTE ADDRESS IN PNTLIS
                        ; THEN ADD 4 TO R2
CLRB    @(R2)+          ; CLEAR BYTE POINTED TO BY SECOND
                        ; ABSOLUTE ADDRESS IN PNTLIS
                        ; THEN ADD 4 TO R2
MOVL    R10,@(R0)+      ; MOVE R10 TO LOCATION WHOSE ADDRESS
                        ; IS POINTED TO BY R0; THEN ADD 4
                        ; TO R0
```

4.1.5 Autodecrement Mode

In autodecrement mode, the processor decrements the contents of the register by the size of the operand data type; then the register contains the address of the operand. The processor decrements the register by 1, 2, 4, 8, or 16 for byte, word, longword, quadword, or octaword operands, respectively.

Autodecrement mode can be used with index mode (see Section 4.3), but the index register cannot be the same as the register specified in autodecrement mode.

Formats

```
-(Rn)
-(AP)
-(FP)
-(SP)
```

n
    A number in the range of 0 through 12.

Example

```
CLRO    -(R1)           ; SUBTRACT 8 FROM R1 AND ZERO THE
                        ; OCTAWORD WHOSE ADDRESS IS THEN
                        ; IN R1
MOVZBL  R3,-(SP)        ; PUSH THE ZERO-EXTENDED LOW BYTE
                        ; OF R3 ONTO THE STACK AS A LONGWORD
                        ; ONTO THE STACK
CMPB    R1,-(R0)        ; SUBTRACT 1 FROM R0 AND COMPARE LOW
                        ; BYTE OF R1 WITH BYTE WHOSE ADDRESS
                        ; IS NOW IN R0
```

4.1.6 Displacement Mode

In displacement mode, the sum of the contents of the register and the displacement (sign extended to a longword) is the address of the operand.

Displacement mode can be used with index mode (see Section 4.3).

Formats

```
dis(Rn)
dis(AP)
dis(FP)
dis(SP)
```

n

A number in the range of 0 through 12.

dis

An expression specifying a displacement; the expression can be preceded by one of the following displacement length specifiers, which indicate the number of bytes needed to store the displacement.

Displacement Length Specifier	Meaning
B^	Displacement requires 1 byte
W^	Displacement requires 1 word (2 bytes)
L^	Displacement requires 1 longword (4 bytes)

If no displacement length specified precedes the expression and the value of the expression is known, the assembler chooses the smallest number of bytes (1, 2, or 4) needed to store the displacement. If no length specifier precedes the expression and the value of the expression is unknown, the assembler reserves 1 word (2 bytes) for the displacement. Note that if the displacement is either relocatable or defined later in the source program, the assembler considers it unknown. If the actual displacement does not fit in the memory reserved, the linker displays an error message.

Example

```
MOVAB   KEYWORDS,R3       ; GET ADDRESS OF KEYWORDS
MOVB    B^IO(R3),R4       ; GET BYTE WHOSE ADDRESS IS
                          ; IO PLUS ADDRESS OF KEYWORDS
                          ; THE DISPLACEMENT IS STORED AS A BYTE
MOVB    B^ACCOUNT(R3),R5; GET BYTE WHOSE ADDRESS IS ACCOUNT
                          ; PLUS ADDRESS OF KEYWORDS
                          ; THE DISPLACEMENT IS STORED AS A BYTE
CLRW    L^STA(R1)         ; CLEAR WORD WHOSE ADDRESS
                          ; IS STA PLUS CONTENTS OF R1
                          ; THE DISPLACEMENT IS STORED
                          ; AS A LONGWORD
MOVL    R0,-2(R2)         ; MOVE R0 TO ADDRESS THAT IS -2
                          ; PLUS THE CONTENTS OF R2
                          ; THE DISPLACEMENT IS STORED AS A BYTE
TSTB    EXTRN(R3)         ; TEST THE BYTE WHOSE ADDRESS
                          ; IS EXTRN PLUS THE
                          ; ADDRESS OF KEYWORDS
                          ; THE DISPLACEMENT IS STORED AS A WORD
                          ; SINCE EXTRN IS UNDEFINED
MOVAB   2(R5),R0          ; MOVE <CONTENTS OF R5> + 2
                          ; TO R0
```

Note

If the value of the displacement is 0 and no displacement length is specified, the assembler uses register deferred mode rather than displacement mode.

4.1.7 Displacement Deferred Mode

In displacement deferred mode, the sum of the contents of the register and the displacement (sign extended to a longword) is the address of the operand address (a pointer to the operand).

Displacement deferred mode can be used with index mode (see Section 4.3).

Formats

```
@dis(Rn)
@dis(AP)
@dis(FP)
@dis(SP)
```

n

A number in the range of 0 through 12.

dis

An expression specifying a displacement; the expression can be preceded by one of the following displacement length specifiers, which indicate the number of bytes needed to store the displacement.

Displacement Length Specifier	Meaning
B^	Displacement requires 1 byte
W^	Displacement requires 1 word (2 bytes)
L^	Displacement requires 1 longword (4 bytes)

If no displacement length specifier precedes the expression and the value of the expression is known, the assembler chooses the smallest number of bytes (1, 2, or 4) needed to store the displacement. If no length specifier precedes the expression and the value of the expression is unknown, the assembler reserves 1 word (2 bytes) for the displacement. Note that if the displacement is either relocatable or defined later in the source program, the assembler considers it unknown. If the actual displacement does not fit in the memory reserved, the linker displays an error message.

Example

```
MOVAL   ARRPOINT,R6        ; GET ADDRESS OF ARRAY OF POINTERS
CLRL    @16(R6)            ; CLEAR LONGWORD POINTED TO BY
                           ; LONGWORD WHOSE ADDRESS IS 16
                           ; PLUS THE ADDRESS OF ARRPOINT
                           ; THE DISPLACEMENT IS STORED AS A BYTE
MOVL    @B^OFFS(R6),@RSOFF(R6) ; MOVE THE LONGWORD POINTED TO
                           ; BY LONGWORD WHOSE ADDRESS IS
                           ; OFFS PLUS THE ADDRESS OF ARRPOINT
                           ; TO THE ADDRESS POINTED TO BY
                           ; LONGWORD WHOSE ADDRESS IS
                           ; RSOFFS PLUS THE ADDRESS OF ARRPOINT
                           ; THE FIRST DISPLACEMENT IS STORED AS A BYTI
                           ; THE SECOND DISPLACEMENT IS STORED AS A WOI
CLRW    @84(R2)            ; CLEAR THE WORD THAT IS POINTED
                           ; TO BY LONGWORD AT 84 PLUS THE
                           ; CONTENTS OF R2--THE ASSEMBLER USES
                           ; BYTE DISPLACEMENT AUTOMATICALLY
```

4.1.8 Literal Mode

In literal mode, the value of the literal is stored in the addressing mode byte itself.

Formats

```
#literal
S^#literal
```

literal

An expression, an integer constant, or a floating-point constant. The literal must fit in the short literal form. That is, integers must be in the range of 0 through 63 and floating-point constants must be one of the 64 values listed in Table 4-2. Floating-point short literals are stored with a 3-bit exponent and a 3-bit fraction. Table 4-2 also shows the value of the exponent and the fraction for each literal. See the VAX-11 Architecture Handbook for information on the format of short literals.

Table 4-2
Floating Point Short Literals

Exponent \ Fraction	0	1	2	3	4	5	6	7
0	0.5	0.5625	0.625	0.6875	0.75	0.8125	0.875	0.9375
1	1.0	1.125	1.25	1.37	1.5	1.625	1.75	1.875
2	2.0	2.25	2.5	2.75	3.0	3.25	3.5	3.75
3	4.0	4.5	5.0	5.5	6.0	6.5	7.0	7.5
4	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0
5	16.0	18.0	20.0	22.0	24.0	26.0	28.0	30.0
6	32.0	36.0	40.0	44.0	48.0	52.0	56.0	60.0
7	64.0	72.0	80.0	88.0	96.0	104.0	112.0	120.0

Example

```
MOVL    #1,R0           ; R0 IS SET TO 1; THE 1 IS STORED
                        ; IN THE INSTRUCTION AS A SHORT
                        ; LITERAL
MOVB    S^#CR,R1        ; THE LOW BYTE OF R1 IS SET
                        ; TO THE VALUE CR
                        ; CR IS STORED IN THE INSTRUCTION
                        ; AS A SHORT LITERAL
                        ; IF CR IS NOT IN RANGE 0-63,
                        ; THE LINKER PRODUCES A TRUNCATION
                        ; ERROR
MOVF    #0.625,R6       ; R6 IS SET TO THE FLOATING
                        ; POINT VALUE 0.625; IT IS STORED
                        ; IN THE FLOATING POINT SHORT
                        ; LITERAL FORM
```

Notes

1. When the #literal format is used, the assembler chooses whether to use literal mode or immediate mode (see Section 4.2.4). The assembler uses immediate mode if any of the following conditions are met:

   - The value of the literal does not fit in the short literal form
   - The literal is a relocatable or external expression (see Section 3.5)
   - The literal is an expression that contains undefined symbols

   The difference between immediate mode and literal mode is the amount of storage that it takes to store the literal in the instruction.

2. The S^#literal format forces the assembler to use literal mode.

4.2 PROGRAM COUNTER MODES

The program counter modes use PC for a general register.

There are five program counter modes:

- Relative
- Relative Deferred
- Absolute
- Immediate
- General

4.2.1 Relative Mode

In relative mode, the address specified is the address of the operand. The assembler stores the address as a displacement from PC.

Relative mode can be used with index mode (see Section 4.3).

Format

    address

address
: An expression specifying an address; the expression can be preceded by one of the following displacement length specifiers, which indicate the number of bytes needed to store the displacement.

Displacement Length Specifier	Meaning
B^	Displacement requires 1 byte
W^	Displacement requires 1 word (2 bytes)
L^	Displacement requires 1 longword (4 bytes)

If no displacement length specifier precedes the address expression and the value of the expression is known, the assembler chooses the smallest number of bytes (1, 2, or 4) needed to store the displacement. If no length specifier precedes the address expression and the value of the expression is unknown, the assembler uses the default displacement length (see the description of .DEFAULT in Chapter 5). If the address expression is either defined later in the program or defined in another program section, the assembler considers the value unknown.

Example

```
MOVL    LABEL,R1          ; GET LONGWORD AT LABEL; THE
                          ; ASSEMBLER USES DEFAULT
                          ; DISPLACEMENT UNLESS LABEL
                          ; PREVIOUSLY DEFINED IN THIS SECTION
CMPL    W^<DATA+4>,R10    ; COMPARE R10 WITH LONGWORD AT
                          ; ADDRESS DATA+4;  THE ASSEMBLER
                          ; USES A WORD DISPLACEMENT
```

4.2.2 Relative Deferred Mode

In relative deferred mode, the address specified is the address of the operand address (a pointer to the operand). The assembler stores the address specified as a displacement from PC.

Relative deferred mode can be used with index mode (see Section 4.3).

Format

@address

address
An expression specifying an address; the expression can be preceded by one of the following displacement length specifiers, which indicate the number of bytes needed to store the displacement.

Displacement Length Specifier	Meaning
B^	Displacement requires 1 byte
W^	Displacement requires 1 word (2 bytes)
L^	Displacement requires 1 longword (4 bytes)

If no displacement length specifier precedes the address expression and the value of the expression is known, the assembler chooses the smallest number of bytes (1, 2, or 4) needed to store the displacement. If no length specifier precedes the address expression and the value of the expression is unknown, the assembler uses the default displacement length (see the description of .DEFAULT in Chapter 5). If the address expression is either defined later in the program or defined in another program section, the assembler considers the value unknown.

Example

```
        CLRL    @W^PNTR         ; CLEAR LONGWORD POINTED TO BY
                                ; LONGWORD AT PNTR; THE ASSEMBLER
                                ; USES A WORD DISPLACEMENT
        INCB    @L^COUNTS+4     ; INCREMENT BYTE POINTED TO BY
                                ; LONGWORD AT COUNTS+4; ASSEMBLER
                                ; USES A LONGWORD DISPLACEMENT
```

4.2.3 Absolute Mode

In absolute mode, the address specified is the address of the operand. The address is stored as an absolute virtual address (compare relative mode, where the address is stored as a displacement from PC).

Absolute mode can be used with index mode (see Section 4.3).

Format

@#address

address
An expression specifying an address.

Example

```
        CLRL    @#^X1100        ; CLEAR THE CONTENTS OF LOCATION 1100(HEX)
        CLRB    @#ACCOUNT       ; CLEAR THE CONTENTS OF LOCATION
                                ; ACCOUNT; THE ADDRESS IS STORED
                                ; ABSOLUTELY, NOT AS A DISPLACEMENT
        CALLS   #3,@#SYS$FAO    ; CALL THE PROCEDURE SYS$FAO WITH
                                ; THREE ARGUMENTS ON THE STACK
```

4.2.4 Immediate Mode

In immediate mode, the literal specified is the operand.

Formats

#literal
I^#literal

literal
An expression, an integer constant, or a floating-point constant.

Example

```
        MOVL    #1000,R0        ; R0 IS SET TO 1000; THE OPERAND 1000
                                ; IS STORED IN A LONGWORD
        MOVB    #BAR,R1         ; THE LOW BYTE OF R1 IS SET
                                ; TO THE VALUE OF BAR
        MOVF    #0.1,R6         ; R6 IS SET TO THE FLOATING
                                ; POINT VALUE 0.1; IT IS STORED
                                ; AS A 4-BYTE FLOATING POINT
                                ; VALUE (IT CAN NOT BE
                                ; REPRESENTED AS A SHORT LITERAL)
        ADDL2   I^#5,R0         ; THE 5 IS STORED IN A LONGWORD
                                ; BECAUSE THE I^ FORCES THE
                                ; ASSEMBLER TO USE IMMEDIATE MODE;
```

Notes

1. When the #literal format is used, the assembler chooses whether to use literal mode (Section 4.1.8) or immediate mode. If the literal is an integer from 0 through 63 or a floating-point constant that fits in the short literal form, the assembler uses literal mode. If the literal is an expression, the assembler uses literal mode if all the following conditions are met:

   - The expression is absolute
   - The expression contains no undefined symbols
   - The value of the expression fits in the short literal form

   In all other cases, the assembler uses immediate mode.

   The difference between immediate mode and literal mode is the amount of storage required to store the literal in the instruction. The assembler stores an immediate mode literal in a byte, word, or longword depending on the operand data type.

2. The I^#literal format forces the assembler to use immediate mode.

4.2.5 General Mode

In general mode, the address specified is the address of the operand. The linker converts the addressing mode to either relative or absolute mode. If the address is relocatable, the linker converts general mode to relative mode. If the address is absolute, the linker converts general mode to absolute mode. General mode is used to write position-independent code when the programmer does not know whether the address is relocatable or absolute. A general addressing mode operand requires 5 bytes of storage.

General mode can be used with index mode (see Section 4.3).

Format

G^address

address
: An expression specifying an address.

Example

```
        CLRL    G^LABEL_1         ; CLEARS THE LONGWORD AT LABEL_1
                                  ; IF LABEL_1 IS DEFINED AS ABSOLUTE
                                  ; THEN THIS IS CONVERTED TO ABSOLUTE
                                  ; MODE; IF IT IS DEFINED AS
                                  ; RELOCATABLE, THEN THIS IS CONVERTED
                                  ; TO RELATIVE MODE
        CALLS   #5,G^SYS$SERVICE  ; CALLS PROCEDURE SYS$SERVICE
                                  ; WITH 5 ARGUMENTS ON STACK
```

4.3 INDEX MODE

Index mode is a general register mode that can be used only in combination with another mode, called the base mode. The base mode can be any addressing mode except register, immediate, literal, index, or branch. The assembler first evaluates the base mode to get the base address. Then the assembler adds the base address to the product of the contents of the index register and the number of bytes of the operand data type. This sum is the operand address.

Combining index mode with the other addressing modes produces the following addressing modes:

- Register Deferred Index
- Autoincrement Index
- Autoincrement Deferred Index
- Autodecrement Index
- Displacement Index
- Displacement Deferred Index
- Relative Index
- Relative Deferred Index
- Absolute Index
- General Index

The process of first evaluating the base mode and then adding the index register is the same for each of these modes.

Formats

```
base-mode[Rx]
base-mode[AP]
base-mode[FP]
base-mode[SP]
```

base-mode
: Any addressing mode except register, immediate, literal, index, or branch, specifying the base address.

x
: A number in the range 0 through 12, specifying the index register.

Table 4-3 lists the formats of index mode addressing.

Examples

```
;
; REGISTER DEFERRED INDEX MODE
;
OFFS=20                             ; DEFINE OFFS
        MOVAB   BLIST,R9            ; GET ADDRESS OF BLIST
        MOVL    #OFFS,R1            ; SET UP INDEX REGISTER
        CLRB    (R9)[R1]            ; CLEAR BYTE WHOSE ADDRESS
                                    ; IS THE ADDRESS OF BLIST
                                    ; PLUS 20*1
        CLRQ    (R9)[R1]            ; CLEAR QUADWORD WHOSE
                                    ; ADDRESS IS THE ADDRESS
                                    ; OF BLIST PLUS 20*8
        CLRO    (R9)[R1]            ; CLEAR OCTAWORD WHOSE
                                    ; ADDRESS IS THE ADDRESS
                                    ; OF BLIST PLUS 20*16
;
; AUTOINCREMENT INDEX MODE
;
        CLRW    (R9)+[R1]           ; CLEAR WORD WHOSE ADDRESS
                                    ; IS ADDRESS OF BLIST PLUS
                                    ; 20*2; R9 NOW CONTAINS
                                    ; ADDRESS OF BLIST+2
;
; AUTOINCREMENT DEFERRED INDEX MODE
;
        MOVAL   POINT,R8            ; GET ADDRESS OF POINT
        MOVL    #30,R2              ; SET UP INDEX REGISTER
        CLRW    @(R8)+[R2]          ; CLEAR WORD WHOSE ADDRESS
                                    ; IS 30*2 PLUS THE ADDRESS
                                    ; STORED IN POINT; R8 NOW
                                    ; CONTAINS 4 PLUS ADDRESS OF
                                    ; POINT
;
; DISPLACEMENT DEFERRED INDEX MODE
;
        MOVAL   ADDARR,R9           ; GET ADDRESS OF ADDRESS ARRAY
        MOVL    #100,R1             ; SET UP INDEX REGISTER
        TSTF    @40(R9)[R1]         ; TEST FLOATING POINT VALUE
                                    ; WHOSE ADDRESS IS 100*4 PLUS
                                    ; THE ADDRESS STORED AT (ADDARR+40)
```

Table 4-3
Index Mode Addressing

Mode	Format*
Register Deferred Index	(Rn)[Rx]
Autoincrement Index	(Rn)+[Rx]
Autoincrement Deferred Index	@(Rn)+[Rx]
Autodecrement Index	-(Rn)[Rx]
Displacement Index	dis(Rn)[Rx]
Displacement Deferred Index	@dis(Rn)[Rx]
Relative Index	address[Rx]
Relative Deferred Index	@address[Rx]
Absolute Index	@#address[Rx]
General Index	G^address[Rx]

* Key:

Rn
Any general register R0 through R12 or the AP, FP, or SP register.

Rx
Any general register R0 through R12 or the AP, FP, or SP register. Rx cannot be the same register as Rn in the autoincrement index, autoincrement deferred index, and decrement index addressing modes.

dis
An expression specifying a displacement.

address
An expression specifying an address.

Notes

1. If the base mode alters the contents of its register (autoincrement, autoincrement deferred, and autodecrement), the index mode cannot specify the same register.

2. The index register is added to the address after the base mode is completely evaluated. For example, in autoincrement deferred index mode, the base register contains the address of the operand address. The index register (times the length of the operand data type) is added to the operand address rather than to the address stored in the base register.

4.4 BRANCH MODE

In branch mode, the address is stored as an implied displacement from PC. This mode can only be used in branch instructions. The displacement for conditional branch instructions and the BRB instruction is stored in a byte. The displacement for the BRW instruction is stored in a word (2 bytes). A byte displacement allows a range of 127 bytes forward and 128 bytes backward. A word displacement allows a range of 32767 bytes forward and 32768 bytes backward. The displacement is relative to the updated PC, the byte past the byte or word where the displacement is stored. See the VAX-11 Architecture Handbook for more information on the branch instructions.

Format

address

address
: An expression that represents an address.

Example

```
ADDL3   (R1)+,R0,TOTAL  ; TOTAL VALUES AND SET CONDITION
                        ; CODES
BLEQ    LABEL1          ; BRANCH TO LABEL1 IF RESULT IS
                        ; LESS THAN OR EQUAL TO 0
BRW     LABEL           ; BRANCH UNCONDITIONALLY TO LABEL
```

CHAPTER 5

GENERAL ASSEMBLER DIRECTIVES

The general assembler directives provide facilities for performing eleven types of functions. Table 5-1 lists these types of functions and the directives that fall under them. The remainder of this chapter describes the directives in detail, showing their formats and giving examples of their use. For ease of reference, the directives are presented in alphabetical order in this chapter. In addition, Appendix B contains a summary of all assembler directives.

Table 5-1
Summary of General Assembler Directives

Category	Directives[1]
Listing Control Directives	.SHOW (.LIST) .NOSHOW (.NLIST) .TITLE .SUBTITLE (.SBTTL) .IDENT .PAGE
Message Display Directives	.PRINT .WARN .ERROR
Assembler Option Directives	.ENABLE (.ENABL) .DISABLE (.DSABL) .DEFAULT
Data Storage Directives	.BYTE .WORD .LONG .ADDRESS .QUAD .OCTA .PACKED .ASCII .ASCIC .ASCID .ASCIZ .F_FLOATING (.FLOAT) .D_FLOATING (.DOUBLE) .G_FLOATING .H_FLOATING .SIGNED_BYTE .SIGNED_WORD

1. The alternate form, if any, is given in parentheses.

(continued on next page)

Table 5-1 (Cont.)
Summary of General Assembler Directives

Category	Directives[1]
Location Control Directives	.ALIGN .EVEN .ODD .BLKA .BLKB .BLKD .BLKF .BLKG .BLKH .BLKL .BLKO .BLKQ .BLKW .END
Program Sectioning Directives	.PSECT .SAVE_PSECT (.SAVE) .RESTORE_PSECT (.RESTORE)
Symbol Control Directives Directives	.GLOBAL (.GLOBL) .EXTERNAL (.EXTRN) .DEBUG .WEAK
Routine Entry Point Definition Directives	.ENTRY .TRANSFER .MASK
Conditional and Subconditional Assembly Block Directives	.IF .ENDC .IF_FALSE (.IFF) .IF_TRUE (.IFT) .IF_TRUE_FALSE (.IFTF) .IIF
Cross-Reference Directives	.CROSS .NOCROSS
Instruction Generation Directives	.OPDEF .REF1 .REF2 .REF4 .REF8 .REF16

1. The alternate form, if any, is given in parentheses.

.ADDRESS

.ADDRESS -- ADDRESS STORAGE DIRECTIVE

.ADDRESS stores successive longwords containing addresses in the object module. DIGITAL recommends that .ADDRESS rather than .LONG be used for storing address data to provide additional information to the linker. In shareable images, addresses specified with .ADDRESS produce position-independent code. See the VAX-11 MACRO User's Guide for a discussion of specifying addresses in position-independent code.

Format

    .ADDRESS address-list

Parameter

address-list

> A list of symbols or expressions, separated by commas, that VAX-11 MACRO interprets as addresses. Repetition factors are not allowed.

Example

```
TABLE:  .ADDRESS   LAB_4,LAB_3,ROUTTERM    ;  REFERENCE TABLE
```

.ALIGN

.ALIGN -- LOCATION COUNTER ALIGNMENT DIRECTIVE

.ALIGN aligns the location counter to the boundary specified by either an integer or a keyword.

Formats

```
.ALIGN integer[,expression]
.ALIGN keyword[,expression]
```

Parameters

integer

An integer in the range of 0 through 9. The location counter is aligned at an address that is the value of 2 raised to the power of the integer.

keyword

One of five keywords that specify the alignment boundary. The location counter is aligned to an address that is the next multiple of the values listed below.

Keyword	**Size (in Bytes)**
BYTE	2^0 = 1
WORD	2^1 = 2
LONG	2^2 = 4
QUAD	2^3 = 8
PAGE	2^9 = 512

expression

Specifies the fill value to be stored in each byte. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5).

Example

```
.ALIGN  BYTE,0                  ; BYTE ALIGNMENT-FILL WITH NULL
.ALIGN  WORD                    ; WORD ALIGNMENT
.ALIGN  3,^A/ /                 ; QUAD ALIGNMENT-FILL WITH BLANKS
.ALIGN  PAGE                    ; PAGE ALIGNMENT
```

Notes

1. The alignment specified in .ALIGN cannot exceed the alignment of the program section in which the alignment is attempted (see the description of .PSECT). For example, if the default program section alignment (BYTE) is being used and .ALIGN is specified with a WORD or larger alignment, the assembler displays an error message.

2. If the optional expression is supplied, the bytes skipped by the location counter (if any) are filled with the value of that expression. Otherwise, the bytes are zero filled.

3. Although most instructions do not require any data alignment other than byte alignment, execution speed is improved by the following alignments:

Data Length	Alignment
Word	Word
Longword	Longword
Quadword	Quadword

.ASCIx

.ASCIx -- ASCII CHARACTER STORAGE DIRECTIVES

VAX-11 MACRO has four ASCII character storage directives:

Directive	Function
ASCII	ASCII string storage
ASCIC	Counted ASCII string storage
ASCID	String-descriptor ASCII string storage
ASCIZ	Zero-terminated ASCII string storage

Each directive is followed by a string of characters enclosed in a pair of matching delimiters. The delimiters can be any printable character except the space, tab, equal sign (=), semicolon (;), or left angle bracket (<). The character used as the delimiter cannot appear in the string itself. Alphanumeric characters can be used as delimiters; however, nonalphanumeric characters should be used to avoid confusion.

Any character except the null, carriage return, and form feed characters can appear within the string. The assembler does not convert lowercase alphabetic characters to uppercase.

ASCII character storage directives convert the characters to their 8-bit ASCII value (see Appendix A) and store them one character to a byte.

Any character, including the null, carriage return, and form feed characters, can also be represented by an expression enclosed in angle brackets outside of the delimiters. You must define the ASCII values of null, carriage return, and form feed with a direct assignment statement. The ASCII character storage directives store the 8-bit binary value specified by the expression.

ASCII strings can be continued over several lines but the string on each line must be delimited at both ends; however, a different pair of delimiters can be used for each line. For example:

```
CR=13
LF=10

        .ASCII  /ABC DEFG/
        .ASCIZ  @Any character can be delimiter@
        .ASCIC  ? lowercase is not converted to UPPER?
        .ASCII  ? this is a test!?<CR><LF>!Isn't it?!
        .ASCII  \ Angle Brackets <are part <of> this> string \
        .ASCII  / This string is continued / -
                \ on the next line \
        .ASCII  <CR><LF>! this string includes an expression! -

                <128+CR>? whose value is a 13 plus 128?
```

The following sections describe each of the four ASCII character storage directives, giving the formats and examples of each.

.ASCII

.ASCII -- ASCII STRING STORAGE DIRECTIVE

.ASCII stores in the next available byte the ASCII value of each character in the ASCII string or the value of each byte expression.

Format

.ASCII string

Parameter

string

A delimited ASCII string.

Example

```
CR=13                                   ; DIRECT ASSIGNMENT STATEMENTS
LF=10                                   ; DEFINE CR AND LF
        .ASCII  "DATE:  17-NOV-1979"    ; DELIMITER IS "
        .ASCII  /EOF/<CR><LF>           ; DELIMITER IS /
```

.ASCIC

.ASCIC -- COUNTED ASCII STRING STORAGE DIRECTIVE

.ASCIC performs the same function as .ASCII, except that .ASCIC inserts a count byte before the string data. The count byte contains the length of the string in bytes. The length given includes any bytes of nonprintable characters outside the delimited string but excludes the count byte.

.ASCIC is useful in copying text because the count indicates the length of the text to be copied.

Format

.ASCIC string

Parameter

string

A delimited ASCII string.

Example

```
CR=13                               ; DIRECT ASSIGNMENT STATEMENT DEFINES CR

        .ASCIC  #HELLO#<CR>         ; THIS COUNTED ASCII STRING
                                    ; IS EQUIVALENT TO
        .BYTE   6                   ; THE COUNT
        .ASCII  #HELLO#<CR>         ; FOLLOWED BY THE ASCII STRING
```

.ASCID

.ASCID -- STRING-DESCRIPTOR ASCII STRING STORAGE DIRECTIVE

.ASCID performs the same function as ASCII, except that .ASCID inserts a string descriptor before the string data. The string descriptor consists of 1) two bytes that specify the length of the string, 2) two bytes of descriptor information, and 3) a longword containing a position-independent pointer to the string. String descriptors are used in calling procedures (see Appendix C of the VAX-11 Architecture Handbook). Position-independence is discussed in the VAX-11 MACRO User's Guide.

Format

.ASCID string

Parameter

string

A delimited ASCII string.

Example

```
DESCR1: .ASCID  /ARGUMENT FOR CALL/      ;  STRING DESCRIPTOR
DESCR2: .ASCID  /SECOND ARGUMENT/        ;  ANOTHER ONE
          .
          .
          .
        PUSHAL  DESCR1                   ;  PUT ADDRESS OF DESCRIPTORS
        PUSHAL  DESCR2                   ;  ON THE STACK
        CALLS   #2,STRNG_PROC            ;  CALL PROCEDURE
```

.ASCIZ

.ASCIZ -- ZERO-TERMINATED ASCII STRING STORAGE DIRECTIVE

.ASCIZ performs the same function as .ASCII, except that .ASCIZ appends a null byte as the final character of the string. Thus, when a list or text string is created with an .ASCIZ directive, the user need only perform a search for the null character in the last byte to determine the end of the string.

Format

.ASCIZ string

Parameter

string

A delimited ASCII string.

Example

```
FF=12                                    ; DEFINE FF

        .ASCIZ  /ABCDEF/                 ; 6 CHARACTERS IN STRING
                                         ; 7 BYTES OF DATA
        .ASCIZ  /A/<FF>/B/               ; 3 CHARACTERS IN STRINGS
                                         ; 4 BYTES OF DATA
```

.BLKx -- BLOCK STORAGE ALLOCATION DIRECTIVES

VAX-11 MACRO has ten block storage directives:

Directive	Function
.BLKA	Reserves storage for addresses (longwords)
.BLKB	Reserves storage for byte data
.BLKD	Reserves storage for double-precision, floating-point data (quadwords)
.BLKF	Reserves storage for single-precision, floating-point data (longwords)
.BLKG	Reserves storage for G_floating data (quadwords)
.BLKH	Reserves storage for H_floating data (octawords)
.BLKL	Reserves storage for longword data
.BLKO	Reserves storage for octaword data
.BLKQ	Reserves storage for quadword data
.BLKW	Reserves storage for word data

Each directive reserves storage for a different data type. The value of the expression determines the number of data items for which VAX-11 MACRO reserves storage. For example, .BLKL 4 reserves storage for 4 longwords of data and .BLKB 2 reserves storage for 2 bytes of data.

The total number of bytes reserved is equal to the length of the data type times the value of the expression as follows:

Directive	Number of Bytes Allocated
.BLKB	Value of expression
.BLKW	2 * value of expression
.BLKA .BLKF .BLKL	4 * value of expression
.BLKD .BLKG .BLKQ	8 * value of expression
.BLKH .BLKO	16 * value of expression

Formats

```
.BLKA expression
.BLKB expression
.BLKD expression
.BLKF expression
.BLKG expression
.BLKH expression
.BLKL expression
.BLKO expression
.BLKQ expression
.BLKW expression
```

Parameter

expression

An expression specifying the amount of storage to be allocated. All the symbols in the expression must be defined and the expression must be an absolute expression (see Section 3.5). If the expression is omitted, a default value of 1 is assumed.

Example

```
.BLKB   15                  ; SPACE FOR 15 BYTES
.BLKO   3                   ; SPACE FOR 3 OCTAWORDS (48 BYTES)
.BLKL   1                   ; SPACE FOR 1 LONGWORD (4 BYTES)
.BLKF   <3*4>               ; SPACE FOR 12 SINGLE PRECISION
                            ; FLOATING-POINT VALUES (48 BYTES)
```

.BYTE

.BYTE -- BYTE STORAGE DIRECTIVE

.BYTE generates successive bytes of binary data in the object module.

Format

```
.BYTE expression-list
```

Parameter

expression-list

One or more expressions separated by commas. Each expression is first evaluated as a longword expression. Then the value of each expression is truncated to 1 byte. The value of each expression should be in the range of 0 through 255 for unsigned data or in the range of -128 through +127 for signed data.

Each expression optionally can be followed by a repetition factor delimited by square brackets. An expression followed by a repetition factor has the format:

```
expression1[expression2]
```

expression1

An expression that specifies the value to be stored.

[expression2]

An expression that specifies the number of times the value will be repeated. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5). The square brackets are required.

Example

```
.BYTE   <1024-1000>*2           ; STORES A VALUE OF 48
.BYTE   ^XA,FIF,10,65-<21*3>    ; STORES 4 BYTES OF DATA
.BYTE   0                       ; STORES 1 BYTE OF DATA
.BYTE   X,X+3[5*4],Z            ; STORES 22 BYTES OF DATA
```

Notes

1. The assembler displays an error message if the high-order 3 bytes of the longword expression has a value other than 0 or ^XFFFFFF.

2. At link time, a relocatable expression can result in a value that exceeds 1 byte. In this case, the VAX-11 Linker issues a truncation diagnostic message for the object module in question. For example:

```
A:      .BYTE  A            ; RELOCATABLE VALUE 'A' WILL
                            ; CAUSE VAX-11 LINKER TRUNCATION
                            ; DIAGNOSTIC IF THE STATEMENT
                            ; HAS A VIRTUAL ADDRESS OF 256
                            ; OR ABOVE
```

3. The .SIGNED_BYTE directive is the same as .BYTE except the assembler displays a diagnostic message if a value in the range from 128 to 255 is specified. See the description of .SIGNED_BYTE for more information.

.CROSS
.NOCROSS

.CROSS AND .NOCROSS -- CROSS-REFERENCE DIRECTIVES

VAX-11 MACRO produces a cross-reference listing when the CROSS qualifier is specified in the MACRO command. The .CROSS and .NOCROSS directives control which symbols are included in the cross-reference listing. The .CROSS and .NOCROSS directives have an effect only if /CROSS_REFERENCE was specified in the MACRO command (see the VAX-11 MACRO User's Guide).

By default, the cross-reference listing includes the definition and all the references to every symbol in the module. The cross-reference listing can be disabled for all symbols or for a specified list of symbols.

.NOCROSS without a symbol list disables the cross-reference listing of all symbols. .CROSS without a symbol list reenables the cross-reference listing. Any symbol definition or reference that appears after .NOCROSS without a symbol list and before the next .CROSS with no argument list is excluded from the cross reference listing.

.NOCROSS with a symbol list disables the cross-reference listing for the listed symbols. .CROSS with a symbol list reenables the cross-reference listing of the listed symbols.

Formats

```
.CROSS
.CROSS symbol-list
.NOCROSS
.NOCROSS symbol-list
```

Parameter

symbol-list

A list of legal symbol names separated by commas.

Examples

```
        .NOCROSS                  ; STOP CROSS REFERENCE
LAB1:   MOVL      LOC1,LOC2       ; COPY DATA
        .CROSS                    ; REENABLE CROSS REFERENCE
```

The definition of LAB1 and the references to LOC1 and LOC2 are not included in the cross reference listing.

```
        .NOCROSS  LOC1            ; DO NOT CROSS REFERENCE LOC1
LAB2:   MOVL      LOC1,LOC2       ; COPY DATA
        .CROSS    LOC1            ; REENABLE CROSS REFERENCE
                                  ; OF LOC1
```

The definition of LAB2 and the reference to LOC2 are included in the cross reference, but the reference to LOC1 is not included in the cross reference.

Notes

1. .CROSS without a symbol list will not reenable the cross-reference listing of a symbol specified in .NOCROSS with a symbol list.

2. If the cross-reference listing of all symbols is disabled, .CROSS with a symbol list will have no effect until the cross-reference listing is reenabled by .CROSS without a symbol list.

.DEBUG

.DEBUG -- DEBUG SYMBOL ATTRIBUTE DIRECTIVE

.DEBUG specifies that the symbols in the list are made known to the debugger. During an interactive debugging session, these symbols can be used to refer to memory locations or to examine the values assigned to the symbols.

Format

```
.DEBUG symbol-list
```

Parameter

symbol-list

A list of legal symbols separated by commas.

Example

```
.DEBUG  INPUT,OUTPUT,-              ; MAKE THESE SYMBOLS KNOWN
        LAB_30,LAB_40               ; TO THE DEBUGGER
```

Note

The assembler adds the symbols in the symbol list to the symbol table in the object module. The programmer need not specify global symbols in the .DEBUG directive because global symbols automatically are put in the object moduule's symbol table. See the description of .ENABLE for information on making information about all symbols available to the debugger.

.DEFAULT

.DEFAULT -- DEFAULT CONTROL DIRECTIVE

.DEFAULT determines the default displacement length for the relative and relative deferred addressing modes (see Sections 4.2.1 and 4.2.2).

Format

```
.DEFAULT     DISPLACEMENT, keyword
```

Parameter

keyword

One of three keywords--BYTE, WORD, LONG--indicating the default displacement length.

Example

```
.DEFAULT    DISPLACEMENT,WORD     ; WORD IS DEFAULT
MOVL    LABEL,R1                  ; ASSEMBLER USES WORD
                                  ; DISPLACEMENT UNLESS
                                  ; LABEL HAS BEEN DEFINED
.DEFAULT    DISPLACEMENT,LONG     ; LONG IS DEFAULT
INCB    @COUNTS+4                 ; ASSEMBLER USES LONGWORD
                                  ; DISPLACEMENT UNLESS
                                  ; COUNTS HAS BEEN DEFINED
```

Notes

1. .DEFAULT has no effect on the default displacement for displacement and displacement deferred addressing modes (see Sections 4.1.6 and 4.1.7).

2. If there is no .DEFAULT in a source module, the default displacement length for the relative and relative deferred addressing modes is a longword.

.D_FLOATING
.DOUBLE

.D_FLOATING -- FLOATING POINT STORAGE DIRECTIVE

.D_FLOATING evaluates the specified floating-point constants and stores the results in the object module. .D_FLOATING generates 64-bit, double-precision, floating-point data (1 bit of sign, 8 bits of exponent, and 55 bits of fraction). See the description of .F_FLOATING for information on storing single precision floating point numbers and the descriptions of .G_FLOATING and .H_FLOATING for descriptions of other floating point numbers.

Format

```
.D_FLOATING literal-list
.DOUBLE literal-list
```

Parameter

literal-list

A list of floating-point constants (see Section 3.2.2). The constants cannot contain any unary or binary operators except unary plus or unary minus.

Example

```
.D_FLOATING  1000,1.0E3,1.0000000E-9    ; CONSTANT
.DOUBLE     3.1415928, 1.107153423828   ; LIST
.D_FLOATING  5, 10, 15, 0, 0.5          ;
```

Notes

1. Double precision floating point numbers are always rounded. They are not affected by .ENABLE TRUNCATION.
2. The floating point constants in the literal list must not be preceded by the floating point operator (^F).

.DISABLE

.DISABLE -- FUNCTION CONTROL DIRECTIVE

.DISABLE disables, or inhibits, the specified assembler functions. See the description of .ENABLE for more information.

Format

.DISABLE argument-list

Parameter

argument-list

One or more of the symbolic arguments listed in Table 5-2 in the description of .ENABLE. Either the long form or the short form of the symbolic arguments can be used. If multiple arguments are specified, they must be separated by commas, spaces, or tabs.

Note

The alternate form of .DISABLE is .DSABL.

.ENABLE

.ENABLE -- FUNCTION CONTROL DIRECTIVE

.ENABLE enables the specified assembly function. .ENABLE and its negative form, .DISABLE, control the following assembler functions.

- Creating local label blocks.
- Making all local symbols available to the debugger and enabling the traceback feature.
- Specifying that undefined symbol references are external references.
- Truncating or rounding single-precision floating-point numbers.
- Suppressing the listing of symbols that are defined but not referenced.
- Specifying that all PC references are absolute not relative.

Format

.ENABLE argument-list

Parameter

argument-list

One or more of the symbolic arguments listed in Table 5-2. Either the long form or the short form of the symbolic arguments can be used.

If multiple arguments are specified, they must be separated by commas, spaces, or tabs.

Table 5-2
.ENABLE and .DISABLE Symbolic Arguments

Long Form	Short Form	Default Condition	Function
ABSOLUTE	AMA	Disabled	When ABSOLUTE is enabled, all PC relative addressing modes are assembled as absolute addressing modes.
DEBUG	DBG	Disabled	When DEBUG is enabled, all local symbols are included in the object module's symbol table for use by the debugger.
GLOBAL	GBL	Enabled	When GLOBAL is enabled, all undefined symbols are considered external symbols. When GLOBAL is disabled, any undefined symbol that is not listed in a .EXTERNAL directive causes an assembly error.
LOCAL_BLOCK	LSB	Disabled	When LOCAL_BLOCK is enabled, the current local label block is ended and a new one is started. When LOCAL_BLOCK is disabled, the current local label block is ended. See Section 3.4 for a complete description of local label blocks.
SUPPRESSION	SUP	Disabled	When SUPPRESSION is enabled, all symbols that are defined but not referred to are not listed in the symbol table. When SUPPRESSION is disabled, all symbols that are defined are listed in the symbol table.
TRACEBACK	TBK	Enabled	When TRACEBACK is enabled, the program section names and lengths, module names, and routine names are included in the object module for use by the debugger. When TRACEBACK is disabled, VAX-11 MACRO excludes this information and, in addition, does not make any local symbol information available to the debugger.

(continued on next page)

Table 5-2 (Cont.)
.ENABLE and .DISABLE Symbolic Arguments

Long Form	Short Form	Default Condition	Function
TRUNCATION	FPT	Disabled	When TRUNCATION is enabled, single-precision floating-point numbers are truncated. When TRUNCATION is disabled, single-precision floating-point numbers are rounded. D_floating, G_floating, and H_floating numbers are not affected by .ENABLE TRUNCATION; they are always rounded.

Example

```
        .ENABLE ABSOLUTE, GLOBAL          ; ASSEMBLE RELATIVE ADDRESS MODE
                                          ; AS ABSOLUTE ADDRESS MODE.
                                          ; UNDEFINED REFERENCES ARE GLOBAL

        .DISABLE TRUNCATION,TRACEBACK     ; ROUND FLOATING-POINT NUMBERS.
                                          ; DO NOT PUT ANY DEBUGGING
                                          ; INFORMATION INTO OBJECT MODULE
```

Note

The alternate form of .ENABLE is .ENABL.

.END

.END -- ASSEMBLY TERMINATION DIRECTIVE

.END terminates the source program. No additional text should occur beyond this point in the current source file or in any additional source files specified in the command line for this assembly. If any additional text does occur, the assembler ignores the text. The additional text does not appear in either the listing file or the object file.

Format

```
.END [symbol]
```

Parameter

symbol

The address (called the transfer address) at which program execution is to begin.

Example

```
.ENTRY  START,0              ; ENTRY MASK
.                            ; MAIN PROGRAM
.
.
.END    START
```

Notes

1. The transfer address must be in a program section that has the EXE attribute (see the description of .PSECT).

2. When an executable image consisting of several object modules is linked, only one object module should be terminated by an .END directive that specifies a transfer address. All other object modules should be terminated by .END directives that do not specify a transfer address. If an executable image either contains no transfer address or contains more than one transfer address, the VAX-11 Linker displays an error message.

3. If the source program contains an unterminated conditional code block when the .END directive is specified, the assembler displays an error message.

.ENDC

.ENDC -- END CONDITIONAL DIRECTIVE

.ENDC terminates the conditional range started by the .IF directive. See the description of .IF for more information and examples.

Format

```
.ENDC
```

.ENTRY

.ENTRY -- ENTRY DIRECTIVE

.ENTRY defines a symbolic name for an entry point and stores a register save mask (2 bytes) at that location. The symbol is defined as a global symbol with a value equal to the value of the location counter at the .ENTRY directive. The entry point can be used as the transfer address of the program. The register save mask is used to determine which registers are saved before the procedure is called. These saved registers are automatically restored when the procedure returns control to the calling program. See the description of the procedure call instructions in the VAX-11 Architecture Handbook.

Format

.ENTRY symbol,expression

Parameter

symbol

The symbolic name for the entry point.

expression

The register save mask for the entry point. The expression must be an absolute expression and must not contain any undefined symbols.

Example

```
.ENTRY  CALC,^M<R2,R3,R7>        ; PROCEDURE STARTS HERE.
                                 ; REGISTERS 2,3,7 ARE
                                 ; PRESERVED BY CALL AND
                                 ; RET INSTRUCTIONS
```

Notes

1. The register mask operator (^M) is convenient to use for setting the bits in the register save mask (see Section 3.6.2.2).

2. An assembly error occurs if the expression has bits 0, 1, 12, or 13 set. These bits correspond to the registers R0, R1, AP, and FP and are reserved for the CALL interface.

3. DIGITAL recommends that .ENTRY be used to define all callable entry points including the transfer address of the program. Although the following construct also defines an entry point, its use is discouraged:

   ```
   symbol:: .WORD   expression
   ```

   Although a procedure starting with this construct can be called, the entry mask is not checked for any illegal registers and the symbol cannot be used in a .MASK directive.

4. .ENTRY should be used only for procedures that will be called by the CALLS or CALLG instruction. A routine that is entered by the BSB or JSB instruction should not use .ENTRY because these instructions do not expect a register save mask. These routines should begin in the following format:

```
symbol::  first instruction
```

The first instruction of the routine immediately follows the symbol.

.ERROR

.ERROR -- ERROR DIRECTIVE

.ERROR causes the assembler to display an error message on the terminal or batch log file and in the listing file (if there is one).

Format

```
.ERROR [expression] ;  comment
```

Parameters

expression

An expression whose value is displayed when .ERROR is encountered during assembly.

; comment

A comment that is displayed when .ERROR is encountered during assembly. The comment must be preceded by a semicolon.

Example

```
.IF DEFINED      LONG_MESS
.IF GREATER      1000-WORK_AREA
.ERROR 25        ; NEED LARGER WORK_AREA
.ENDC
.ENDC
```

If the symbol LONG_MESS is defined and if the symbol WORK_AREA has a value of 1000 or less, the following error message is displayed:

```
%MACRO-E-GENERR, Generated ERROR:  25 NEED LARGER WORK_AREA
```

Notes

1. .ERROR, .WARN, and .PRINT are called the message display directives. They can be used to display information indicating that a macro call contains an error or an illegal set of conditions (see Chapter 6 for more information on macro calls).

2. When the assembly is finished, the assembler displays the total number of errors, warnings, and information messages, and the sequence numbers of the lines causing the errors or warnings on the terminal. See the VAX-11 MACRO User's Guide for more information on errors and warnings.

3. If .ERROR is included in a macro library (see the VAX-11 MACRO User's Guide), the comment should end with an additional semicolon. Otherwise, the librarian will strip the comment from the directive and it will not be displayed when the macro is called.

4. The line containing the .ERROR directive is not included in the listing file.

5. If the expression has a value of 0, it is not displayed in the error message.

.EVEN

.EVEN -- EVEN LOCATION COUNTER ALIGNMENT DIRECTIVE

.EVEN ensures that the current value of the location counter is even by adding 1 if the current value is odd. If the current value is already even, no action is taken.

Format

```
.EVEN
```

.EXTERNAL

.EXTERNAL -- EXTERNAL SYMBOL ATTRIBUTE DIRECTIVE

.EXTERNAL indicates that specified symbols are external; that is, the symbols are defined in another object module and cannot be defined until link time (see Section 3.3.3).

Format

```
.EXTERNAL symbol-list
```

Parameter

symbol-list

A list of legal symbols separated by commas.

Example

```
.EXTERNAL      SIN,TAN,COS       ; THESE SYMBOLS ARE DEFINED IN
.EXTERNAL      SINH,COSH,TANH    ; EXTERNALLY ASSEMBLED MODULES
```

Notes

1. If the GLOBAL argument is enabled (see Table 5-2 in the description of .ENABLE), all unresolved references will be marked as global and external. Thus, if GLOBAL is enabled, the programmer need not specify .EXTERNAL. However, if GLOBAL is disabled, the programmer must explicitly specify .EXTERNAL to declare any symbols that are defined externally but referred to in the current module.

2. If GLOBAL is disabled and the assembler finds symbols that are not defined in the current module and are not listed in a .EXTERNAL directive, the assembler displays an error message.

3. The alternate form of .EXTERNAL is .EXTRN.

.F_FLOATING
.FLOAT

.F_FLOATING (.FLOAT) -- FLOATING-POINT STORAGE DIRECTIVE

.F_FLOATING evaluates the specified floating-point constants and stores the results in the object module. .F_FLOATING generates 32-bit, single-precision, floating-point data (1 bit of sign, 8 bits of exponent, and 23 bits of fractional significance). See the description of .D_FLOATING for information on storing double-precision floating-point numbers and the descriptions of .G_FLOATING and .H_FLOATING for descriptions of other floating point numbers.

Format

```
.F_FLOATING literal-list
.FLOAT literal-list
```

Parameter

literal-list

A list of floating-point constants (see Section 3.2.2). The constants cannot contain any unary or binary operators except unary plus and unary minus.

Example

```
.F_FLOATING   134.5782,74218.34E20       ; SINGLE PRECISION
.F_FLOATING   134.2,0.1342E3,1342E-1     ; THESE ALL GENERATE 134.2
.F_FLOATING   -0.75,1E38,-1.0E-37        ; DATA
.FLOAT        0,25,50                    ; LIST
```

Notes

1. See the description of .ENABLE for information on specifying floating-point rounding or truncation.

2. The floating point constants in the literal list must not be preceded by the floating point unary operator (^F).

.G_FLOATING

.G_FLOATING -- G_FLOATING POINT STORAGE DIRECTIVE

G_FLOATING evaluates the specified floating-point constants and stores the results in the object module. .G_FLOATING generates 64-bit data (1 bit of sign, 11 bits of exponent, and 52 bits of fraction).

Format

```
.G_FLOATING literal-list
```

Parameters

literal-list

A list of G_floating point constants (see Section 3.2.2). The constants cannot contain any unary or binary operators except unary plus or unary minus.

Example

```
.G_FLOATING     1000,  1.0E3,  1.0000000E-9     ;constant list
```

Notes

1. G_floating point numbers are always rounded. They are not affected by the .ENABLE TRUNCATION directive.

2. The floating point constants in the literal list must not be preceded by the floating point operator (^F).

.GLOBAL

.GLOBAL -- GLOBAL SYMBOL ATTRIBUTE DIRECTIVE

.GLOBAL indicates that specified symbol names are either globally defined in the current module or externally defined in another module (see Section 3.3.3).

Format

```
.GLOBAL symbol-list
```

Parameter

symbol-list

A list of legal symbol names separated by commas.

Example

```
.GLOBAL LAB_40,LAB_30           ; MAKE THESE SYMBOL NAMES
                                ; GLOBALLY KNOWN
.GLOBAL UKN_13                  ; TO ALL LINKED MODULES
```

Notes

1. .GLOBAL is provided for MACRO-11 compatibility only. DIGITAL recommends that global definitions be specified by a double colon or double equals sign (see Section 2.2.1 and 3.8) and that external references be specified by .EXTERNAL (when necessary).

2. The alternate form of .GLOBAL is .GLOBL.

.H_FLOATING

.H_FLOATING -- H_FLOATING POINT STORAGE DIRECTIVE

H_FLOATING evaluates the specified floating-point constants and stores the results in the object module. .H_FLOATING generates 128-bit data (1 bit of sign, 15 bits of exponent, and 112 bits of fraction).

Format

```
.H_FLOATING literal-list
```

Parameters

literal-list

A list of H_floating point constants (see Section 3.2.2). The constants cannot contain any unary or binary operators except unary plus or unary minus.

Example

```
.H_FLOATING     36912,  15.0E18,  1.0000000E-9     ;constant list
```

Notes

1. H_floating point numbers are always rounded. They are not affected by the .ENABLE TRUNCATION directive.

2. The floating point constants in the literal list must not be preceded by the floating point operator (^F).

.IDENT

.IDENT -- IDENTIFICATION DIRECTIVE

.IDENT provides a means of identifying the object module. This identification is in addition to the name assigned to the object module with .TITLE. A character string can be specified in .IDENT to label the object module. This string is printed in the header of the listing file as well as appearing in the object module.

Format

```
.IDENT string
```

Parameter

string

A 1- to 31-character string that identifies the module, such as a string that specifies a version number. The string must be delimited. The delimiters can be any paired printing characters, other than the left angle bracket (<) or the semicolon (;), as long as the delimiting character is not contained in the text string itself.

Example

```
.IDENT  /3-47/                     ; VERSION AND EDIT NUMBERS
```

The character string 3-47 is included in the object module.

Notes

1. If one source module contains more than one .IDENT, the last directive given establishes the character string that forms part of the object module identification.

2. If the delimiting characters do not match, or if an illegal delimiting character is used, the assembler displays an error message.

.IF

.IF -- CONDITIONAL ASSEMBLY BLOCK DIRECTIVES

A conditional assembly block is a series of source statements that is assembled only if a certain condition is met. .IF starts the conditional block and .ENDC ends the conditional block. Each .IF must have a corresponding .ENDC. The .IF directive contains a condition test and one or two arguments. The condition test specified is applied to the argument(s). If the test is met, all MACRO statements between .IF and .ENDC are assembled. If the test is not met, the statements are not assembled. An exception to this occurs when subconditional directives are used (see the description of .IF_x directive).

Conditional blocks can be nested, that is a conditional block can be inside of another conditional block. In this case the statements in the inner conditional block are assembled only if the condition is met for both the outer and inner block.

Format

```
.IF condition argument(s)
  .
  .
  .
range
  .
  .
  .
.ENDC
```

Parameters

condition

A specified condition that must be met if the block is to be included in the assembly. Table 5-3 lists the conditions that can be tested by the conditional assembly directives. The condition must be separated from the argument(s) by a comma, space, or tab.

argument(s)

The symbolic argument(s) or expression(s) of the specified conditional test. If the argument is an expression, it cannot contain any undefined symbols and must be an absolute expression (see Section 3.5).

range

The block of source code that is conditionally included in the assembly.

Table 5-3
Condition Tests for Conditional Assembly Directives

Condition Test		Complement Condition Test		Argument Type	Number of Arguments	Condition that Assembles Block
Long Form	Short Form	Long Form	Short Form			
EQUAL	EQ	NOT_EQUAL	NE	Expression	1	Expression is equal to 0 /not equal to 0
GREATER	GT	LESS_EQUAL	LE	Expression	1	Expression is greater than 0 /less than or equal to 0
LESS_THAN	LT	GREATER_EQUAL	GE	Expression	1	Expression is less than 0 /greater than or equal to 0
DEFINED	DF	NOT_DEFINED	NDF	Symbolic	1	Symbol is defined /not defined
BLANK[1]	B	NOT_BLANK[1]	NB	Macro	1	Argument is blank /nonblank
IDENTICAL[1]	IDN	DIFFERENT[1]	DIF	Macro	2	Arguments are identical /different

1. The BLANK, NOT_BLANK, IDENTICAL, and DIFFERENT conditions are only useful in macro definitions. Chapter 6 describes macro directives in detail.

Examples

1. An example of a conditional assembly directive is:

```
.IF EQUAL  ALPHA+1        ; ASSEMBLE BLOCK IF ALPHA+1=0
  .                       ; DO NOT ASSEMBLE IF ALPHA+1 NOT=0
  .
  .
.ENDC
```

2. Nested conditional directives take the form:

```
.IF    condition,argument(s)
.IF    condition,argument(s)
  .
  .
  .
.ENDC
.ENDC
```

3. The following conditional directives can govern whether assembly is to occur:

```
.IF DEFINED  SYM1
.IF DEFINED  SYM2
  .
  .
  .
.ENDC
.ENDC
```

In this example, if the outermost condition is not satisfied, no deeper level of evaluation of nested conditional statements within the program occurs. Therefore, both SYM1 and SYM2 must be defined for the code to be assembled.

Notes

1. If .ENDC occurs outside a conditional assembly block, the assembler displays an error message.

2. VAX-11 MACRO permits a nesting depth of 31 conditional assembly levels. If a statement attempts to exceed this nesting level depth, the assembler displays an error message.

3. The assembler displays an error message if .IF specifies any of the following: a condition test other than those in Table 5-3, an illegal argument, or a null argument specified in an .IF directive.

4. The .SHOW and .NOSHOW directives control whether condition blocks that are not assembled are included in the listing file.

.IF_x

.IF_x -- SUBCONDITIONAL ASSEMBLY BLOCK DIRECTIVES

VAX-11 MACRO has three subconditional assembly block directives:

Directive	Function
.IF_FALSE	If the condition of the assembly block tests false, the program is to include the source code following the .IF_FALSE directive and continuing up to the next subconditional directive or to the end of the conditional assembly block.
.IF_TRUE	If the condition of the assembly block tests true, the program is to include the source code following the .IF_TRUE directive and continuing up to the next subconditional directive or to the end of the conditional assembly block.
.IF_TRUE_FALSE	Always include the source code following the .IF_TRUE_FALSE directive and continuing up to the next subconditional directive or to the end of the conditional assembly block. This source code is included regardless of whether the condition of the assembly block tests true or false.

The implied argument of a subconditional directive is the condition test specified when the conditional assembly block was entered. A conditional or subconditional directive in a nested conditional assembly block is not evaluated if the preceding (or outer) condition in the block is not satisfied (see examples 3 and 4 below).

A conditional block with a subconditional directive is different than a nested conditional block. If the condition in the .IF is not met, the inner conditional block(s) are not assembled, but a subconditional directive can cause a block to be assembled.

Formats

```
.IF_FALSE
.IF_TRUE
.IF_TRUE_FALSE
```

Examples

1. Assume that symbol SYM is defined:

```
.IF DEFINED   SYM             ; TESTS TRUE SINCE SYM IS DEFINED.
  .                           ; ASSEMBLES THE FOLLOWING CODE.
  .
  .
.IF_FALSE                     ; TESTS FALSE SINCE PREVIOUS
  .                           ; .IF WAS TRUE. DO NOT
  .                           ; ASSEMBLE THE FOLLOWING CODE.
  .
  .
.IF_TRUE                      ; TESTS TRUE. SYM IS DEFINED.
  .                           ; ASSEMBLES THE FOLLOWING CODE.
  .
  .
.IF_TRUE_FALSE                ; ASSEMBLES FOLLOWING CODE
  .                           ; UNCONDITIONALLY.
  .
  .
.IF_TRUE                      ; TESTS TRUE. SYM IS DEFINED.
  .                           ; ASSEMBLES REMAINDER OF
  .                           ; CONDITIONAL ASSEMBLY BLOCK.
  .
.ENDC
```

2. Assume that symbol X is defined and that symbol Y is not defined:

```
.IF DEFINED  X                ; TESTS TRUE. SYMBOL X IS DEFINED.
.IF DEFINED  Y                ; TESTS FALSE. SYMBOL Y IS NOT
                              ; DEFINED.
.IF_FALSE                     ; TESTS TRUE. SYMBOL Y IS NOT
  .                           ; DEFINED.
  .                           ; ASSEMBLES THE FOLLOWING CODE.
  .
  .
.IF_TRUE                      ; TESTS FALSE. SYMBOL Y IS NOT
  .                           ; DEFINED.
  .                           ; DOES NOT ASSEMBLE THE FOLLOWING
  .                           ; CODE.
  .
  .
.ENDC
.ENDC
```

3. Assume that symbol A is defined and that symbol B is not defined:

```
.IF DEFINED  A                ; TESTS TRUE. A IS DEFINED.
                              ; ASSEMBLES THE FOLLOWING CODE.
  .
  .
  .
.IF_FALSE                     ; TESTS FALSE. A IS DEFINED. DOES
                              ; NOT ASSEMBLE THE FOLLOWING CODE.
  .
  .
  .
.IF NOT_DEFINED B             ; NESTED CONDITIONAL DIRECTIVE
  .                           ; IS NOT EVALUATED.
  .
  .
.ENDC
.ENDC
```

4. Assume that symbol X is not defined but symbol Y is defined:

```
.IF DEFINED  X                 ; TESTS FALSE. SYMBOL X IS NOT
  .                            ; DEFINED.
                               ; DOES NOT ASSEMBLE THE
  .                            ; FOLLOWING CODE.
.IF DEFINED  Y                 ; NESTED CONDITIONAL DIRECTIVE
  .                            ; IS NOT EVALUATED.
  .
  .
.IF_FALSE                      ; NESTED SUBCONDITIONAL
  .                            ; DIRECTIVE IS NOT EVALUATED.
  .
  .
.IF_TRUE                       ; NESTED SUBCONDITIONAL
  .                            ; DIRECTIVE IS NOT EVALUATED.
  .
  .
.ENDC
.ENDC
```

Notes

1. If a subconditional directive appears outside a conditional assembly block, the assembler displays an error message.

2. The alternate forms of .IF_FALSE, .IF_TRUE, and .IF_TRUE_FALSE are .IFF, .IFT, and .IFTF.

.IIF

.IIF -- IMMEDIATE CONDITIONAL ASSEMBLY BLOCK DIRECTIVE

.IIF provides a means of writing a one-line conditional assembly block. The condition to be tested and the conditional assembly block are expressed completely within the line containing the .IIF directive; no terminating .ENDC statement is required.

Format

```
.IIF condition argument(s), statement
```

Parameters

condition

One of the legal condition tests defined for conditional assembly blocks in Table 5-3 (See the description of .IF). The condition must be separated from the argument(s) by a comma, space, or tab.

argument(s)

The argument associated with the immediate conditional directive; that is, an expression or symbolic argument (described in Table 5-3). If the argument is an expression, it cannot contain any undefined symbols and must be an absolute expression (see Section 3.3.3). The argument(s) must be separated from the statement by a comma.

statement

The statement to be assembled if the condition is satisfied.

Example

```
Condition Argument      Statement

.IIF DEFINED EXAM,      BEQL    ALPHA
```

This directive generates the following code if the symbol EXAM is defined within the source program:

```
BEQL    ALPHA
```

Note

The assembler displays an error message if .IIF specifies any of the following: a condition test other than those listed in Table 5-3, an illegal argument, or a null argument.

.LIST

.LIST -- LISTING DIRECTIVE

.LIST is equivalent to the .SHOW. See the description of .SHOW for more information.

Formats

```
.LIST
.LIST argument-list
```

Parameter

argument-list

One or more of the symbolic argument defined in Table 5-7 in the description of .SHOW. Either the long form or the short form of the arguments can be used. If multiple arguments are specified, they must be separated by commas, spaces, or tabs.

.LONG

.LONG -- LONGWORD STORAGE DIRECTIVE

.LONG generates successive longwords (4 bytes) of data in the object module.

Format

```
.LONG expression-list
```

Parameters

expression-list

One or more expressions separated by commas. Each expression optionally can be followed by a repetition factor delimited by square brackets.

An expression followed by a repetition factor has the format:

```
expression1[expression2]
```

expression1

An expression that specifies the value to be stored.

[expression2]

An expression that specifies the number of times the value will be repeated. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5). The square brackets are required.

Example

```
LAB_3:  .LONG   LAB_3,^X7FFFFFFF,^A'ABCD' ; 3 LONGWORDS OF DATA
        .LONG   ^XF@4                     ; 1 LONGWORD OF DATA
        .LONG   0[22]                     ; 22 LONGWORDS OF DATA
```

Note

Each expression in the list must have a value that can be represented in 32 bits.

.MASK

.MASK -- MASK DIRECTIVE

.MASK reserves a word for a register save mask for a transfer vector. See the description of .TRANSFER for more information and for an example of .MASK.

Format

```
.MASK symbol[,expression]
```

Parameters

symbol

A symbol defined in an .ENTRY directive.

expression

A register save mask.

Notes

1. If .MASK does not contain an expression, the assembler directs the linker to copy the register save mask specified in .ENTRY to the word reserved by .MASK.

2. If .MASK contains an expression, the assembler directs the linker to combine this expression with the register save mask specified in .ENTRY and store the result in the word reserved by .MASK. The linker performs an inclusive OR operation to combine the mask in the entry point and the value of the expression. Consequently, a register specified in either .ENTRY or .MASK will be included in the combined mask. See the description of .ENTRY for more information on entry masks.

.NLIST

.NLIST -- LISTING DIRECTIVE

.NLIST is equivalent to .NOSHOW. See the description of .SHOW for more information.

Formats

```
.NLIST
.NLIST argument-list
```

Parameter

argument-list

One or more of the symbolic arguments listed in Table 5-7 in the description of .SHOW. Either the long form or the short form of the arguments can be used. If multiple arguments are specified, they must be separated by commas, spaces, or tabs.

.NOCROSS

.NOCROSS -- CROSS REFERENCE DIRECTIVE

VAX-11 MACRO produces a cross-reference listing when the CROSS qualifier is specified in the MACRO command. The .CROSS and .NOCROSS directives control which symbols are included in the cross-reference listing. The description of .NOCROSS is included with the description of .CROSS.

Formats

```
.NOCROSS
.NOCROSS symbol-list
```

Parameter

symbol-list

A list of legal symbol names separated by commas.

.NOSHOW

.NOSHOW -- LISTING DIRECTIVE

.NOSHOW specifies listing control options. See the description of .SHOW for more information.

Formats

```
.NOSHOW
.NOSHOW argument-list
```

Parameter

argument-list

One or more of the symbolic arguments listed in Table 5-7 in the description of .SHOW. Either the long form or the short form of the arguments can be used. If multiple arguments are specified, they must be separated by commas, spaces, or tabs.

.OCTA

.OCTA -- OCTAWORD STORAGE DIRECTIVE

.OCTA generates 128 bits (16 bytes) of binary data.

Format

```
.OCTA literal
.OCTA symbol
```

Parameters

literal

Any constant value. This value can be preceded by ^O, ^B, ^X, or ^D to specify the radix as octal, binary, hexadecimal, or decimal, respectively; or it can be preceded by ^A to specify ASCII text. Decimal is the default radix.

symbol

A symbol defined elsewhere in the program. This symbol results in a sign-extended, 32-bit value being stored in an octaword.

Example

```
.OCTA    ^A"FEDCBA987654321"      ; EACH ASCII CHARACTER IS STORED
                                  ; IN A BYTE
.OCTA    0                        ; OCTA 0
.OCTA    ^X01234ABCD5678F9        ; OCTA HEX VALUE SPECIFIED
.OCTA    VINTERVAL                ; VINTERVAL HAS A 32 BIT VALUE
                                  ; SIGN EXTENDED
```

Note

.OCTA is like .QUAD and unlike other data storage directives (.BYTE, .WORD, and .LONG) in that it does not evaluate expressions and that it accepts only one value. It does not accept a list.

.ODD

.ODD -- ODD LOCATION COUNTER ALIGNMENT DIRECTIVE

.ODD ensures that the current value of the location counter is odd by adding 1 if the current value is even. If the current value is already odd, no action is taken.

Format

```
.ODD
```

.OPDEF

.OPDEF -- OPCODE DEFINITION DIRECTIVE

.OPDEF defines an opcode, which it inserts into a user-defined opcode table. The assembler searches this table before it searches the permanent symbol table. This directive can redefine an existing opcode name or create a new one.

Format

```
.OPDEF  opcode     value,operand-descriptor-list
```

Parameters

opcode

An ASCII string specifying the name of the opcode. The string can be up to 31 characters long and can contain the letters A through Z; the digits 0 through 9; and the special characters underline (_), dollar sign ($), and period (.). The string should not start with a digit and should not be surrounded by delimiters.

value

An expression that specifies the value of the opcode. The expression must not contain any undefined values and must be an absolute expression (see Section 3.5). The value of the expression must be in the range of 0 through decimal 65535 (hexadecimal FFFF), but the values 252 through 255 cannot be used. The expression is represented as follows:

if 0 ≤ expression ≤ 251	expression is a one-byte opcode.
if expression > 255	expression bits 15:8 are the first byte of the opcode and expression bits 7:0 are the second byte of the opcode.

Values 252 through 255 cannot be used because the architecture specifies these as the start of a two-byte opcode. Note that this representation does not correspond to the representation of these numbers in memory.

operand-descriptor-list

A list of operand descriptors that specifies the number of operands and the type of each. Up to 16 operand descriptors are allowed in the list. Table 5-4 lists the operand descriptors.

Table 5-4
Operand Descriptors

Access Type	Data Type								
	Byte	Word	Long-word	Floating Point	Double Floating Point	G_floating Point	H_floating Point	Quad-word	Octa-word
Address	AB	AW	AL	AF	AD	AG	AH	AQ	AO
Read-only	RB	RW	RL	RF	RD	RG	RH	RQ	RO
Modify	MB	MW	ML	MF	MD	MG	MH	MQ	MO
Write-only	WB	WW	WL	WF	WD	WG	WH	WQ	WO
Field	VB	VW	VL	VF	VD	VG	VH	VQ	VO
Branch	BB	BW	-	-	-	-	-	-	-

Examples

```
.OPDEF   MOVL3    ^XFFA9,RL,ML,WL          ; DEFINES AN
                                  ; INSTRUCTION, MOVL3, WHICH USES
                                  ; THE RESERVED OPCODE FF.
.OPDEF   DIVF2    ^X46,RF,MF      ; REDEFINES THE DIVF2 AND
.OPDEF   MOVC5    ^X2C,RW,AB,AB,RW,AB      ; MOVC5 INSTRUCTIONS.
.OPDEF   CALL     ^X10,BB         ; EQUIVALENT TO A BSBB
```

Notes

1. A macro can also be used to redefine an opcode (see the description of .MACRO in Chapter 6). Note that the macro name table is searched before the user-defined opcode table.

2. .OPDEF is useful in creating "custom" instructions that execute user-written microcode. This directive is supplied to allow programmers to execute their microcode in a MACRO program.

3. The operand descriptors are specified in a format similar to the operand specifier notation described in the VAX-11 Architecture Handbook. The first character specifies the operand access type and the second character specifies the operand data type.

.PACKED

.PACKED -- PACKED DECIMAL STRING STORAGE DIRECTIVE

.PACKED generates packed decimal data, 2 digits per byte. Packed decimal data is useful in calculations requiring exact accuracy. Packed decimal data is operated on by the decimal string instructions. See the VAX-11 Architecture Handbook for more information on the format of packed decimal data.

Format

.PACKED decimal-string[,symbol]

Parameters

decimal-string

A decimal number from 0 through 31 digits long with an optional sign. Each digit can be in the range of 0 through 9 (see Section 3.2.3).

symbol

An optional symbol that is assigned a value equivalent to the number of decimal digits in the string. The sign is not counted as a digit.

Example

```
.PACKED -12,PACK_SIZE             ; PACK_SIZE GETS VALUE OF 2
.PACKED +500
.PACKED 0
.PACKED -0,SUM_SIZE               ; SUM_SIZE GETS VALUE OF 1
```

.PAGE

.PAGE -- PAGE EJECTION DIRECTIVE

.PAGE forces a new page in the listing; the directive itself is not printed in the listing.

VAX-11 MACRO ignores .PAGE in a macro definition. The paging operation is performed only during macro expansion. Chapter 6 describes macro directives and facilities in detail.

Format

.PAGE

.PRINT

.PRINT -- ASSEMBLY MESSAGE DIRECTIVE

.PRINT causes the assembler to display an informational message. The message consists of the value of the expression and the comment specified in the .PRINT directive. The message is displayed on the terminal for interactive jobs and in the log file for batch jobs. The message produced by .PRINT is not considered an error or warning message.

Format

```
.PRINT [expression] ;comment
```

Parameters

expression

An expression whose value is displayed when .PRINT is encountered during assembly.

comment

A comment that is displayed when .PRINT is encountered during assembly. The comment must be preceded by a semicolon.

Example

```
.PRINT  2        ; THE SINE ROUTINE HAS BEEN CHANGED
```

Notes

1. .PRINT, .ERROR, and .WARN are called the message display directives. They can be used to display information indicating that a macro call contains an error or an illegal set of conditions (See Chapter 6 for more information on macro calls).

2. If .PRINT is included in a macro library (see the <u>VAX-11 MACRO User's Guide</u>), the comment should end with an additional semicolon. Otherwise, the comment will be stripped from the directive and will not be displayed when the macro is called.

3. If the expression has a value of 0, it is not displayed with the message.

.PSECT

.PSECT -- PROGRAM SECTIONING DIRECTIVE

.PSECT defines a program section and its attributes and refers to a program section once it is defined.

Program sections can be used to:

- Develop modular programs
- Separate instructions from data
- Allow different modules to access the same data
- Protect read-only data and instructions from being modified
- Identify sections of the object module to the debugger
- Control the order in which program sections are stored in virtual memory

See the VAX-11 MACRO User's Guide for more information on using program sections.

The assembler automatically defines two program sections: the absolute program section and the unnamed (or blank) program section. Any symbol definitions that appear before any instruction, data, or .PSECT directive are placed in the absolute program section. Any instructions or data that appear before the first named program section is defined are placed in the unnamed program section. Any .PSECT directive that does not include a program section name specifies the unnamed program section.

A maximum of 254 user-defined, named program sections can be defined.

When the assembler encounters a .PSECT directive that specifies a new program section name, it creates a new program section and stores the name, attributes, and alignment of the program section. The assembler includes all data and instructions that follow the .PSECT directive in that program section until it encounters another .PSECT directive. The assembler starts all program sections at a location counter of 0 which is relocatable.

If the assembler encounters a .PSECT directive that specifies the name of a previously defined program section, it stores the new data or instructions after the last entry in the previously defined program section. The location counter is set to the value of the location counter at the end of the previously defined program section. The programmer need not list the attributes when continuing a program section but any attributes that are listed must be the same as those previously in effect for the program section. A continuation of a program section cannot contain attributes conflicting with those specified in the original .PSECT directive.

The attributes listed in the .PSECT directive only describe the contents of the program section. The assembler does not check to ensure that the contents of the program section actually include the attributes listed.

However, the assembler and the linker do check that all program sections with the same name have exactly the same attributes. The assembler and linker display an error message if the program section attributes are not consistent.

Program section names are independent of local symbol, global symbol, and macro names. Thus, the same symbolic name can be used for a program section and for a local symbol, global symbol, or macro name.

Formats

```
.PSECT
.PSECT program section-name[,argument-list]
```

Parameters

program-section-name

The name of the program section. This name can be up to 31 characters long and can contain any alphanumeric character and the underline (_), dollar sign ($), and period (.) characters. However, the first character must not be a digit.

argument-list

A list containing the program section attributes and the program section alignment. Table 5-5 lists the attributes and their functions. Table 5-6 lists the default attributes and their opposites. Program sections are aligned when an integer in the range of 0 through 9 is specified or when one of the five keywords listed below is specified. If an integer is specified, the program section is linked to begin at the next virtual address that is a multiple of 2 raised to the power of the integer. If a keyword is specified, the program section is linked to begin at the next virtual address that is a multiple of the values listed below:

Keyword	Size (in Bytes)
BYTE	2^0 = 1
WORD	2^1 = 2
LONG	2^2 = 4
QUAD	2^3 = 8
PAGE	2^9 = 512

BYTE is the default.

Table 5-5
Program Section Attributes

Attribute Name	Function
ABS	Absolute--The linker assigns the program section an absolute address. The contents of the program section can be only symbol definitions (usually definitions of symbolic offsets to data structures that are used by the routines being assembled). An absolute program section contributes no binary code to the image, so its byte allocation request to the linker is 0. The size of the data structure being defined is the size of the absolute program section printed in the "program section synopsis" at the end of the listing. Compare this attribute with its opposite, REL.
CON	Concatenate--Program sections with the same name and attributes (including CON) are merged into one program section. Their contents are merged in the order in which the linker acquires them. The allocated virtual address space is the sum of the individual requested allocations.
EXE	Executable--The program section contains instructions. This attribute provides the capability of separating instructions from read-only and read/write data. The linker uses this attribute in gathering program sections and in verifying that the transfer address is in an executable program section.
GBL	Global--Program sections that have the same name and attributes, including GBL and OVR, will have the same relocatable address in memory even when the program sections are in different clusters (see the VAX-11 Linker Reference Manual for more information on clusters). This attribute is specified for FORTRAN COMMON block program sections (see the VAX-11 FORTRAN User's Guide). Compare this attribute with its opposite, LCL.
LCL	Local--The program section is restricted to its cluster. Compare this attribute with its opposite, GBL.
LIB	Library Segment--Reserved for future use.
NOEXE	Not Executable--The program section contains data only; it does not contain instructions.
NOPIC	Non-Position-Independent Content--The program section is assigned to a fixed location in virtual memory (when it is in a shareable image).
NORD	Nonreadable--Reserved for future use.
NOSHR	No Share--The program section is reserved for private use at execution time by the initiating process.

(continued on next page)

Table 5-5 (Cont.)
Program Section Attributes

Attribute Name	Function
NOWRT	Nonwritable--The program section's contents cannot be altered (written into) at execution time.
OVR	Overlay--Program sections with the same name and attributes, including OVR, have the same relocatable base address in memory. The allocated virtual address space is the requested allocation of the largest overlaying program section. Compare this attribute with its opposite, CON.
PIC	Position-Independent Content--The program section can be relocated; that is, it can be assigned to any memory area (when it is in a shareable image).
RD	Readable--Reserved for future use.
REL	Relocatable--The linker assigns the program section a relocatable base address. The contents of the program section can be code or data. Compare this attribute with its opposite, ABS.
SHR	Share--The program section can be shared at execution time by multiple processes. This attribute is assigned to a program section that can be linked into a shareable image.
USR	User Segment--Reserved for future use.
VEC	Vector-Containing--The program section contains a change mode vector indicating a privileged shareable image. The SHR attribute must be used with VEC.
WRT	Write--The program section's contents can be altered (written into) at execution time.

Table 5-6
Default Program Section Attributes

Default Attribute	Opposite Attribute
CON	OVR
EXE	NOEXE
LCL	GBL
NOPIC	PIC
NOSHR	SHR
RD	NORD
REL	ABS
WRT	NOWRT
NOVEC	VEC

Examples

```
.PSECT   CODE,NOWRT,EXE,LONG        ; PROGRAM SECTION TO CONTAIN
                                    ; EXECUTABLE CODE
.PSECT   RWDATA,WRT,NOEXE,QUAD
                                    ; PROGRAM SECTION TO CONTAIN
                                    ; MODIFIABLE DATA
```

Notes

1. The .ALIGN directive cannot specify an alignment greater than that of the current program section; consequently, .PSECT should specify the largest alignment needed in the program section. For efficiency of execution, an alignment of longword or larger is recommended for all program sections that have longword data.

2. The attributes of the default absolute and the default unnamed program sections are listed below. Note that the program section names include the periods and enclosed spaces.

Program Section Name	Attributes and Alignment
. ABS .	NOPIC,USR,CON,ABS,LCL,NOSHR,NOEXE,NORD,NOWRT,NOVEC,BYTE
. BLANK .	NOPIC,USR,CON,REL,LCL,NOSHR,NOEXE,RD,WRT,NOVEC,BYTE

.QUAD

.QUAD -- QUADWORD STORAGE DIRECTIVE

.QUAD generates 64 bits (8 bytes) of binary data.

Format

```
.QUAD literal
.QUAD symbol
```

Parameters

literal

Any constant value. This value can be preceded by ^O, ^B, ^X, or ^D to specify the radix as octal, binary, hexadecimal, or decimal, respectively; or it can be preceded by ^A to specify the ASCII text operator. Decimal is the default radix.

symbol

A symbol defined elsewhere in the program. This symbol results in a sign-extended, 32-bit value being stored in a quadword.

Example

```
.QUAD   ^A'..ASK?..'            ; EACH ASCII CHARACTER IS STORED
                                ; IN A BYTE
.QUAD   0                       ; QUAD 0
.QUAD   ^X0123456789ABCDEF      ; QUAD HEX VALUE SPECIFIED
.QUAD   ^B1111000111001101      ; QUAD BINARY VALUE SPECIFIED
.QUAD   LABEL                   ; LABEL HAS A 32 BIT
                                ; VALUE ZERO EXTENDED.
```

Note

.QUAD is like .OCTA and different from other data storage directives (.BYTE, .WORD, and .LONG) in that it does not evaluate expressions and that it accepts only one value. It does not accept a list.

.REFn

.REFn -- OPERAND GENERATION DIRECTIVES

VAX-11 MACRO has five operand generation directives used in macros (see Chapter 6) to define new opcodes:

Directive	Function
.REF1	Generates a byte operand
.REF2	Generates a word operand
.REF4	Generates a longword operand
.REF8	Generates a quadword operand
.REF16	Generates an octaword operand

The .REFn directives are provided for compatibility with VAX-11 MACRO V1.0. The .OPDEF directive provides greater functionality and is easier to use than .REFn; consequently, .OPDEF should be used instead of .REFn.

Formats

```
.REF1  operand
.REF2  operand
.REF4  operand
.REF8  operand
.REF16 operand
```

Parameter

operand

An operand of byte, word, longword, quadword, or octaword context, respectively.

Example

```
    .MACRO   MOVL3 A,B,C
    .BYTE    ^XFF,^XA9
    .REF4    A                    ; THIS OPERAND HAS LONGWORD CONTEXT
    .REF4    B                    ; THIS OPERAND HAS LONGWORD CONTEXT
    .REF4    C                    ; THIS OPERAND HAS LONGWORD CONTEXT
    .ENDM    MOVL3

    MOVL3    R0,@LAB-1,(R7)+[R10]
```

This example uses .REF4 to create a new instruction, MOVL3, which uses the reserved opcode FF. See the example in .OPDEF for a preferred method to create a new instruction.

.RESTORE_PSECT

.RESTORE_PSECT -- RESTORE PREVIOUS PROGRAM SECTION CONTEXT DIRECTIVE

.RESTORE_PSECT retrieves the program section from the top of the program section context stack, an internal stack in the assembler. If the stack is empty when .RESTORE_PSECT is issued, the assembler displays an error message. When .RESTORE_PSECT retrieves a program section, it restores the current location counter to the value it had when the program section was saved. The local label block is also restored if it was saved when the program section was saved. See the description of .SAVE_PSECT.

Format

```
.RESTORE_PSECT
```

Example

.RESTORE_PSECT and .SAVE_PSECT are especially useful in macros that define program sections (see Chapter 6). The macro definition below saves the current program section context and defines new program sections. Then, it restores the saved program section. If the macro did not save and restore the program section context each time the macro was invoked, the program section would change.

```
        .MACRO   INITD                        ; INITIALIZE SYMBOLS
                                              ; AND DATA AREAS
        .SAVE_PSECT                           ; SAVE THE CURRENT PSECT
        .PSECT   SYMBOLS,ABS                  ; DEFINE NEW PSECT
HELP_LEV=2                                    ; DEFINE SYMBOLS
MAXNUM=100                                    ; ...
RATE1=16                                      ; ...
RATE2=4                                       ; ...
        .PSECT   DATA,NOEXE,LONG              ; DEFINE ANOTHER PSECT
TABL:   .BLKL    100                          ; 100 LONGWORDS IN TABL
TEMP:   .BLKB    16                           ; MORE STORAGE
        .RESTORE_PSECT                        ; RESTORE THE PSECT
                                              ; IN EFFECT WHEN
                                              ; MACRO IS INVOKED

        .ENDM
```

Note

The alternate form of .RESTORE_PSECT is .RESTORE.

.SAVE_PSECT

.SAVE_PSECT -- SAVE CURRENT PROGRAM SECTION CONTEXT DIRECTIVE

.SAVE_PSECT stores the current program section context on the top of the program section context stack, an internal assembler stack, while leaving the current program section context in effect. The program section context stack can hold 31 entries. Each entry includes the value of the current location counter and the maximum value assigned to the location center in the current program section. If the stack is full when .SAVE_PSECT is encountered, an error occurs.

.SAVE_PSECT and .RESTORE_PSECT are especially useful in macros that define program sections (see Chapter 6). See the description of .RESTORE_PSECT for another example of using .SAVE_PSECT.

Format

```
.SAVE_PSECT [LOCAL_BLOCK]
```

Parameter

LOCAL_BLOCK

An optional keyword that specifies that the current local label is to be saved with the program section context.

Example

MACRO DEFINITION:

```
        .MACRO   ERR_MESSAGE, TEXT         ; SET UP LISTS OF MESSAGES AND
                                           ; POINTERS
        .IIF NOT_DEFINED        MESSAGE_INDEX, MESSAGE_INDEX=0  ;
        .SAVE_PSECT     LOCAL_BLOCK        ; KEEP LOCAL LABELS
        .PSECT   MESSAGE_TEXT              ; LIST OF ERROR MESSAGES
MESSAGE:: .ASCIC /TEXT/
        .PSECT   MESSAGE_POINTERS          ; ADDRESSES OF ERROR MESSAGES
        .ADDRESS        MESSAGE            ; STORE ONE POINTER
        .RESTORE_PSECT                     ; GET BACK LOCAL LABELS
        PUSHL           #MESSAGE_INDEX     ;
        CALLS           #1, PRINT_MESS     ; PRINT MESSAGE
MESSAGE_INDEX=MESSAGE_INDEX+1
        .ENDM    ERR_MESSAGE
```

MACRO CALL:

```
RESETS: CLRL     R4                        ;
        BLBC     R0, 30$          ;
        ERR_MESSAGE     <STRING TOO SHORT>       ; ADD "STRING TOO SHORT"
                                           ; TO LIST OF ERROR MESSAGES
30$:    RSB
```

The use of .SAVE_PSECT LOCAL_BLOCK here means that the local label 30$ is defined in the same local label block as the reference to 30$. If a local label is not defined in the block in which it is referenced, the assembler produces the following error message:

```
%MACRO-E-UNDEFSYM, Undefined Symbol
```

Note

1. The alternate form of .SAVE_PSECT is .SAVE.

.SHOW
.NOSHOW

.SHOW AND .NOSHOW -- LISTING DIRECTIVES

.SHOW and .NOSHOW specify listing control options in the source text of a program. .SHOW and .NOSHOW can be used with or without an argument list.

When used with an argument list, .SHOW causes certain types of lines to be included in the listing file and .NOSHOW causes certain types of lines to be excluded. .SHOW and .NOSHOW control the listing of the source lines that are in conditional assembly blocks (see description of .IF), macros, and repeat blocks (see Chapter 6).

When used without arguments, these directives alter the listing level count. The listing level count is initialized to 0. Each time .SHOW appears in a program, the listing level count is incremented; each time .NOSHOW appears in a program, the listing level count is decremented.

When the listing level count is negative, the listing is suppressed (unless the line contains an error). Conversely, when the listing level count is positive, the listing is generated. When the count is 0, the line is either listed or suppressed, depending on the value of the listing control symbolic arguments.

Formats

```
.SHOW
.SHOW argument-list
.NOSHOW
.NOSHOW argument-list
```

Parameter

argument-list

One or more of the optional symbolic arguments, defined in Table 5-7. Either the long form or the short form of the arguments can be used. Each argument can be used alone or in combination with other arguments. If multiple arguments are specified, they must be separated by commas, tabs, or spaces. If any argument is not specifically included in a listing control statement, its default value (Show or Noshow) is assumed throughout the source program.

Table 5-7
.SHOW and .NOSHOW Symbolic Arguments

Long Form	Short Form	Default	Function
BINARY	MEB	Noshow	Lists macro expansions and repeat block expansions that generate binary code. BINARY is a subset of EXPANSIONS.
CALLS	MC	Show	Lists macro calls and repeat block specifiers.
CONDITIONALS	CND	Show	Lists unsatisfied conditional code associated with the conditional assembly directives.
DEFINITIONS	MD	Show	Lists macro and repeat range definitions that appear in an input source file.
EXPANSIONS	ME	Noshow	Lists macro and repeat range expansions.

Example

```
        .MACRO  XX
          .
          .
          .
        .SHOW                          ; LIST NEXT LINE.
X=.
        .NOSHOW                        ; DO NOT LIST REMAINDER OF MACRO
          .                            ; EXPANSION.
          .
          .
        .ENDM

        .NOSHOW EXPANSIONS             ; DO NOT LIST MACRO EXPANSIONS.
        XX
X=.
```

Notes

1. The listing level count allows macros to be listed selectively; a macro definition can specify .NOSHOW at the beginning to decrement the listing count and can specify .SHOW at the end to restore the listing count to its original value.

2. The alternate forms of .SHOW and .NOSHOW are .LIST and .NLIST.

.SIGNED_BYTE

.SIGNED_BYTE -- SIGNED BYTE DATA DIRECTIVE

.SIGNED_BYTE is equivalent to .BYTE, except that VAX-11 MACRO indicates that the data is signed in the object module. The linker uses this information to test for overflow conditions.

Format

```
.SIGNED_BYTE expression-list
```

Parameters

expression-list

An expression or list of expressions separated by commas. Each expression optionally can be followed by a repetition factor delimited by square brackets.

An expression followed by a repetition factor has the format:

```
expression1[expression2]
```

expression1

An expression that specifies the value to be stored. The value must be in the range -128 to +127.

[expression2]

An expression that specifies the number of times the value will be repeated. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5). The square brackets are required.

Example

```
.SIGNED_BYTE    LABEL1-LABEL2   ;  DATA MUST FIT
.SIGNED_BYTE    ALPHA[20]       ;  IN BYTE
```

Note

Specifying .SIGNED_BYTE allows the linker to detect overflow conditions when the value of the expression is in the range of 128 through 255. Values in this range can be stored as unsigned data but cannot be stored as signed data in a byte.

.SIGNED_WORD

.SIGNED_WORD -- SIGNED WORD STORAGE DIRECTIVE

.SIGNED_WORD is equivalent to .WORD except that the assembler indicates that the data is signed in the object module. The linker uses this information to test for overflow conditions. .SIGNED_WORD is useful after the case instruction to ensure that the displacement fits in a word.

Format

.SIGNED_WORD expression-list

Parameters

expression-list

An expression or list of expressions separated by commas. Each expression optionally can be followed by a repetition factor delimited by square brackets.

An expression followed by a repetition factor has the format:

expression1[expression2]

expression1

An expression that specifies the value to be stored. The value must be in the range -32768 to +32767.

[expression2]

An expression that specifies the number of times the value will be repeated. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5). The square brackets are required.

Example

```
      .MACRO   CASE,SRC,DISPLIST,TYPE=W,LIMIT=#0,NMODE=S^#,?BASE,?MAX
                                         ; MACRO TO USE CASE INSTRUCTION
                                         ; SRC IS SELECTOR, DISPLIST IS LIST
                                         ; OF DISPLACEMENTS, TYPE IS B-BYTE
                                         ; W-WORD, L-LONG, LIMIT IS THE BASE
                                         ; VALUE OF SELECTOR
      CASE'TYPE        SRC,LIMIT,NMODE'<<MAX-BASE>/2>-1
                                         ; CASE INSTRUCTION
BASE:                                    ; LOCAL LABEL SPECIFYING BASE
      .IRP     EP,<DISPLIST>             ; TO SET UP OFFSET LIST
      .SIGNED_WORD     EP-BASE           ; OFFSET LIST
      .ENDR                              ;
MAX:                                     ; LOCAL LABEL USED TO COUNT ARGS
      .ENDM    CASE                      ;

      CASE     IVAR     <ERR_PROC,SORT,REV_SORT>        ; IF IVAR=0, ERROR;
      CASEW    IVAR,#0,S^#<<30001$-30000$>/2>-1
```

```
30000$:                                        ; LOCAL LABEL SPECIFYING BASE
     .SIGNED_WORD     ERR_PROC-30000$                ; OFFSET LIST
     .SIGNED_WORD     SORT-30000$                    ; OFFSET LIST
     .SIGNED_WORD     REV_SORT-30000$                ; OFFSET LIST
30001$:                                        ; LOCAL LABEL USED TO COUNT ARGS
                                               ; =1, FOWARD SORT; =2,BACKWARD SORT

     CASE     TEST     <TEST1,TEST2,TEST3>,L,#1                      ;
     CASEL    TEST,#1,S^#<<30003$-30002$>/2>-1
30002:                                         ; LOCAL LABEL SPECIFYING BASE
     .SIGNED_WORD     TEST1-30002$                   ; OFFSET LIST
     .SIGNED_WORD     TEST2-30002$                   ; OFFSET LIST
     .SIGNED_WORD     TEST3-30002$                   ; OFFSET LIST
30003$:                                        ; LOCAL LABEL USED TO COUNT ARGS
                                               ; VALUE OF TEST CAN BE 1,2, OR 3
```

In this example, the CASE macro uses .SIGNED_WORD to create a CASEB, CASEW, or CASEL instruction. See Chapter 6 for a description of the directives used to define the macro.

Note

> Specifying .SIGNED_WORD allows the linker to detect overflow conditions when the value of the expression is in the range of 32768 through 65535. Values in this range can be stored as unsigned data but cannot be stored as signed data in a word.

.SUBTITLE

.SUBTITLE -- SUBTITLE DIRECTIVE

.SUBTITLE causes the assembler to print a line of text in the table of contents that is produced immediately before the assembly listing. The assembler also prints the line of text as the subtitle on the second line of each assembly listing page. This subtitle text is printed on each page until altered by a subsequent .SUBTITLE directive in the program.

Format

```
.SUBTITLE comment-string
```

Parameter

comment-string

An ASCII string from 1 to 40 characters long; excess characters are truncated. This string represents the line of text to be printed in the table of contents and as the subtitle in the assembly listing.

Example

1. .SUBTITLE CONDITIONAL ASSEMBLY

   This directive causes the assembler to print the following text as the subtitle of the assembly listing:

   ```
   CONDITIONAL ASSEMBLY
   ```

   It also causes the text to be printed out in the listing's table of contents, along with the source page number and the line sequence number of the source statement where .SUBTITLE was specified. The table of contents would have the following format:

2. TABLE OF CONTENTS

   ```
   (1)    5000 ASSEMBLER DIRECTIVES
   (2)    300 MACRO DEFINITIONS
   (2)    2300 DATA TABLES AND INITIALIZATION
   (3)    4800 MAIN ROUTINE
   (4)    2800 CALCULATIONS
   (4)    5000 I/O ROUTINES
   (5)    1300 CONDITIONAL ASSEMBLY
   ```

Note

The alternate form of .SUBTITLE is .SBTTL.

.TITLE

.TITLE -- TITLE DIRECTIVE

.TITLE assigns a name to the object module. This name is the first 31 or fewer nonblank characters following the directive.

Format

.TITLE module-name comment-string

Parameters

module-name

An identifier from 1 to 31 characters long.

comment-string

An ASCII string from 1 to 40 characters long; excess characters are truncated.

Example

```
.TITLE  EVAL    EVALUATES EXPRESSIONS
```

Notes

1. The module name specified with .TITLE bears no relationship to the file specification of the object module, as specified in the VAX-11 MACRO command line. Rather, the object module name appears in the linker load map, and is also the module name that the debugger and librarian recognize.

2. If .TITLE is not specified, MACRO assigns the default name .MAIN. to the object module. If more than one .TITLE directive is specified in the source program, the last .TITLE directive encountered establishes the name for the entire object module.

3. When evaluating the module-name, MACRO ignores all spaces and/or tabs up to the first nonspace/nontab character after .TITLE.

.TRANSFER

.TRANSFER -- TRANSFER DIRECTIVE

.TRANSFER redefines a global symbol for use in a shareable image. The linker redefines the symbol as the value of the location counter at the .TRANSFER directive after a shareable image is linked.

Whenever possible, programs should not need to be relinked when the shareable images to which they are linked change. This can only be achieved if:

- the total size of the shareable image does not change
- the entry points in the shareable image do not change their addresses when the shareable code is changed and the image is relinked.

To avoid changing the size of the shareable image, reserve extra space when first creating the image. To insure that the entry points do not change, create an object module that contains a transfer vector for each entry point and does not change the order of the transfer vectors. Link this object module at the beginning of the shareable image and the addresses will remain fixed even if source code for a routine is changed. After each .TRANSFER directive, a register save mask (for procedures only) and a branch to the first instruction of the routine should appear.

The .TRANSFER directive does not cause any memory to be allocated and does not generate any binary code. It merely generates instructions to the linker to redefine the symbol when a shareable image is being created.

.TRANSFER can be used with procedures entered by the CALLS or CALLG instruction. In this case, .TRANSFER is used with the .ENTRY and .MASK directives. The branch to the actual routine must be a branch to the entry point plus 2. Adding 2 to the address is necessary to bypass the 2-byte register save mask.

Figure 5-1 illustrates the use of transfer vectors.

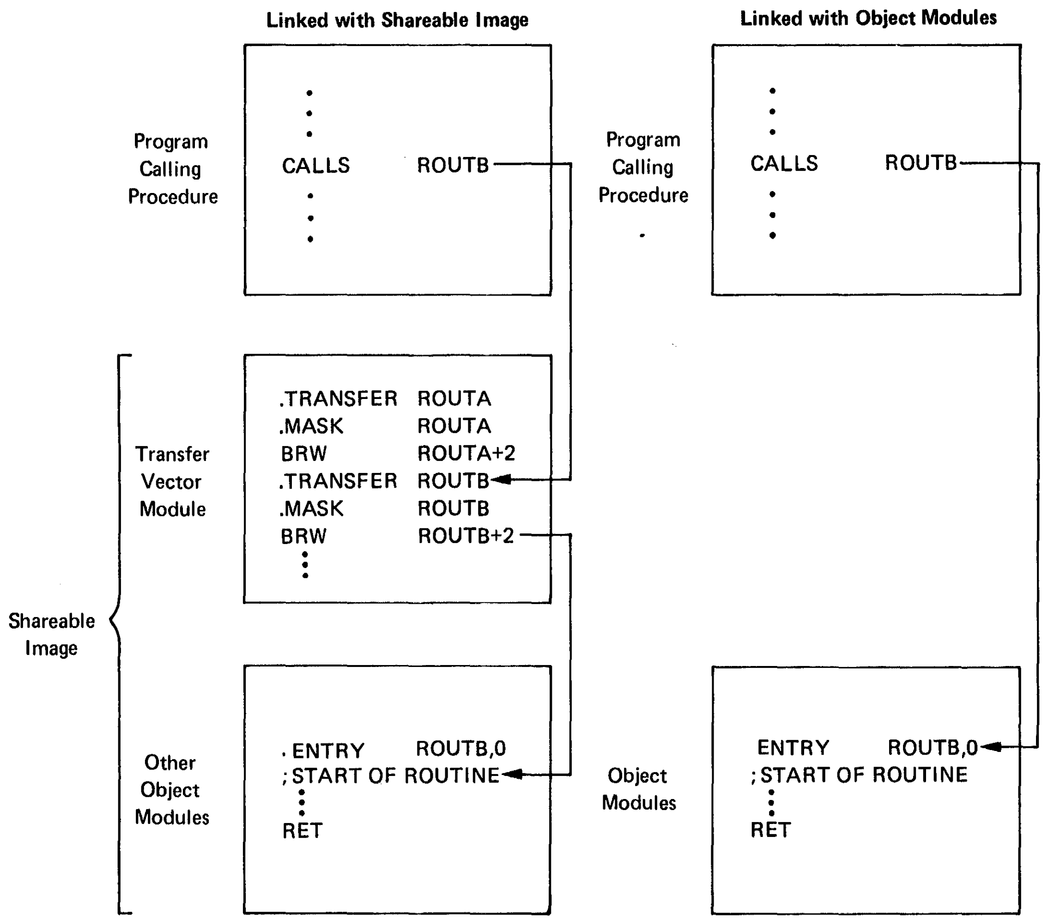

Figure 5-1 Using Transfer Vectors

Format

.TRANSFER symbol

Parameter

symbol

A global symbol that is an entry point in a procedure or routine.

Example

```
.TRANSFER ROUTINE_A
.MASK     ROUTINE_A,^M<R4,R5>    ; COPY ENTRY MASK
                                 ; AND ADD REGISTERS
                                 ; 4 AND 5
BRW       ROUTINE_A+2            ; BRANCH TO ROUTINE
                                 ; (PAST ENTRY MASK)
  .
  .
  .
.ENTRY    ROUTINE_A,^M<R2,R3>    ; ENTRY POINT, SAVE
                                 ; REGISTERS 2 AND 3
  .
  .
  .
RET
```

In this example, .MASK copies a routine's entry mask to the new entry address specified by .TRANSFER. If the routine is placed in a shareable image and then called, registers 2, 3, 4, and 5 will be saved.

.WARN

.WARN -- WARNING DIRECTIVE

.WARN causes the assembler to display a warning message on the terminal or batch log file and in the listing file (if there is one).

Format

```
.WARN [expression] ;comment
```

Parameters

expression

An expression whose value is displayed when .WARN is encountered during assembly.

;comment

A comment that is displayed when .WARN is encountered. The comment must be preceded by a semicolon.

Example

```
.IF DEFINED   FULL
.IF DEFINED   DOUBLE_PREC
.WARN         ; THIS COMBINATION NOT TESTED
.ENDC
.ENDC
```

If the symbols FULL and DOUBLE_PREC are both defined, the following warning message is displayed.

```
%MACRO-W-GENWRN, Generated WARNING:  THIS COMBINATION NOT TESTED
```

Notes

1. .WARN, .ERROR, and .PRINT are called the message display directives. They can be used to display information indicating that a macro call contains an error or an illegal set of conditions (see Chapter 6 for more information on macro calls).

2. When the assembly is finished, the assembler displays the total number of errors, warnings, and information messages, and the page numbers and line numbers of the lines causing the errors or warning on the terminal (or in the batch log file). See the VAX-11 MACRO User's Guide for more information on errors and warnings.

3. If .WARN is included in a macro library (see the VAX-11 MACRO User's Guide), the comment should end with an additional semicolon. Otherwise, the comment will be stripped from the directive and will not be displayed when the macro is called.

4. The line containing the .WARN directive is not included in the listing file.

5. If the expression has a value of 0, it is not displayed in the warning message.

.WEAK

.WEAK -- WEAK SYMBOL ATTRIBUTE DIRECTIVE

.WEAK specifies symbols that are either defined externally in another module or defined globally in the current module. .WEAK suppresses any object library search for the symbol.

When .WEAK specifies a symbol that is not defined in the current module, the symbol is externally defined. If the linker finds the symbol's definition in another module, it uses that definition. If the linker does not find an external definition, the symbol has a value of 0 and the linker does not report an error. The linker does not search a library for the symbol, but if a module brought in from a library for another reason contains the symbol definition, the linker uses it.

When .WEAK specifies a symbol that is defined in the current module, the symbol is considered to be globally defined. However, if this module is inserted in an object library, this symbol is not inserted in the library's symbol table. Consequently, searching the library at link time to resolve this symbol does not cause the module to be included.

Format

```
.WEAK    symbol-list
```

Parameter

symbol-list

A list of legal symbols separated by commas.

Example

```
.WEAK   IOCAR,LAB_3
```

.WORD

.WORD -- WORD STORAGE DIRECTIVE

.WORD generates successive words (2 bytes) of data in the object module.

Format

```
.WORD    expression-list
```

Parameter

expression-list

One or more expressions separated by commas. Each expression optionally can be followed by a repetition factor delimited by square brackets.

An expression followed by a repetition factor has the format:

```
expression1[expression2]
```

expression1

An expression that specifies the value to be stored.

[expression2]

An expression that specifies the number of times the value will be repeated. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5). The square brackets are required.

Example

```
.WORD    ^X3F,FIVE[3],32
```

Notes

1. The expression is first evaluated as a longword, then truncated to a word. The value of the expression should be in the range of -32768 through 32767 for signed data or 0 through 65535 for unsigned data. The assembler displays an error if the high-order 2 bytes of the longword expression have a value other than 0 or ^XFFFF.

2. The .SIGNED_WORD directive is the same as .WORD except that the assembler displays a diagnostic message if a value is in the range from 32768 to 65535.

CHAPTER 6

MACROS

By using macros, a programmer can use a single line to insert a sequence of source lines into a program.

A macro definition contains the source lines of the macro. The macro definition can optionally have formal arguments. These formal arguments can be used throughout the sequence of source lines. Later, the formal arguments are replaced by the actual arguments in the macro call.

The macro call consists of the macro name optionally followed by actual arguments. The assembler replaces the line containing the macro call with the source lines in the macro definition. It replaces any occurrences of formal arguments in the macro definition with the actual arguments specified in the macro call. This process is called the macro expansion.

By default, macro expansions are not printed in the assembly listing. They are printed only when the .SHOW directive (see description in Chapter 5), or the /SHOW qualifier, described in the VAX-11 MACRO User's Guide, specifies the EXPANSIONS argument. In the examples in this chapter, the macro expansions are listed as they would appear if .SHOW EXPANSIONS was specified in the source file or /SHOW EXPANSIONS was specified in the MACRO command string.

The macro directives provide facilities for performing eight categories of functions. Table 6-1 lists these categories and the directives that fall under them. Section 6.1 describes macro arguments. Section 6.2 describes the directives in detail. For ease of reference, the directives are presented in alphabetical order.

Table 6-1
Summary of Macro Directives

Category	Directives[1]
Macro Definition Directives	.MACRO .ENDM
Macro Library Directives	.LIBRARY .MCALL
Macro Deletion Directive	.MDELETE
Macro Exit Directive	.MEXIT
Argument Attribute Directives	.NARG .NCHR .NTYPE
Indefinite Repeat Block Directives	.IRP .IRPC
Repeat Block Directives	.REPEAT (.REPT)
End Range Directive	.ENDR

1. The alternate form, if any, is given in parentheses.

6.1 ARGUMENTS IN MACROS

Macros have two types of arguments: actual and formal. Actual arguments are the strings given in the macro call after the name of the macro. Formal arguments are specified by name in the macro definition: that is, after the macro name in the .MACRO directive. Actual arguments in macro calls and formal arguments in macro definitions can be separated by commas, tabs, or spaces.

The number of actual arguments in the macro call can be less than or equal to the number of formal arguments in the macro definition. But if the number of actual arguments is greater than the number of formal arguments, the assembler displays an error message.

Formal and actual arguments normally maintain a strict positional relationship. That is, the first actual argument in a macro call replaces all occurrences of the first formal argument in the macro definition. However, this strict positional relationship can be overridden by the use of keyword arguments (see Section 6.1.2).

An example of a macro definition using formal arguments follows:

```
.MACRO  STORE   ARG1,ARG2,ARG3
.LONG   ARG1                    ; ARG1 IS FIRST ARGUMENT
.WORD   ARG3                    ; ARG3 IS THIRD ARGUMENT
.BYTE   ARG2                    ; ARG2 IS SECOND ARGUMENT
.ENDM   STORE
```

The following two examples show possible calls and expansions of the macro defined above.

```
1.      STORE   3,2,1                   ; MACRO CALL
        .LONG   3                       ; 3 IS FIRST ARGUMENT
        .WORD   1                       ; 1 IS THIRD ARGUMENT
        .BYTE   2                       ; 2 IS SECOND ARGUMENT

2.      STORE   X,X-Y,Z                 ; MACRO CALL
        .LONG   X                       ; X IS FIRST ARGUMENT
        .WORD   Z                       ; Z IS THIRD ARGUMENT
        .BYTE   X-Y                     ; X-Y IS SECOND ARGUMENT
```

6.1.1 Default Values

Default values are values that are defined in the macro definition. They are used when no value is specified in the macro call for a formal argument.

Default values are specified in the .MACRO directive as follows:

```
formal-argument-name = default-value
```

An example of a macro definition specifying default values follows:

```
        .MACRO  STORE   ARG1=12,ARG2=0,ARG3=1000
        .LONG   ARG1
        .WORD   ARG3
        .BYTE   ARG2
        .ENDM   STORE
```

The following three examples show possible calls and expansions of the macro defined above.

```
1.      STORE                           ; NO ARGUMENTS SUPPLIED
        .LONG   12
        .WORD   1000
        .BYTE   0

2.      STORE   ,5,X                    ; LAST TWO ARGUMENTS SUPPLIED
        .LONG   12
        .WORD   X
        .BYTE   5

3.      STORE   1                       ; FIRST ARGUMENT SUPPLIED
        .LONG   1
        .WORD   1000
        .BYTE   0
```

6.1.2 Keyword Arguments

Keyword arguments allow a macro call to specify the arguments in any order; however, the macro call must specify the same formal argument names that appear in the macro definition. Keyword arguments are useful when a macro definition has many formal arguments, only some of which need to be specified in the call.

In any one macro call the arguments should be either all positional arguments or all keyword arguments. When positional and keyword arguments are combined in a macro, only the positional arguments correspond by position to the formal arguments; the keyword arguments are not used. If a formal argument corresponds to both a positional argument and a keyword argument, the argument that appears last in the macro call overrides any other argument definition for the same argument.

For example, the following macro definition specifies three arguments:

```
.MACRO  STORE   ARG1,ARG2,ARG3
.LONG   ARG1
.WORD   ARG3
.BYTE   ARG2
.ENDM   STORE
```

The following macro call specifies keyword arguments:

```
STORE   ARG3=27+5/4,ARG2=5,ARG1=SYMBL
.LONG   SYMBL
.WORD   27+5/4
.BYTF   5
```

Because the keywords are specified in the macro call, the arguments in the macro call need not be given in the order they were listed in the macro definition.

6.1.3 String Arguments

If an actual argument is a string containing characters that the assembler interprets as separators (such as a tab, space, or comma), the string must be enclosed by delimiters. String delimiters are usually paired angle brackets (<>). However, the assembler also interprets any character after an initial circumflex (^) as a delimiter. Thus, to pass an angle bracket as part of a string, the programmer can use the circumflex form of the delimiter.

The following are examples of delimited macro arguments:

```
<HAVE THE SUPPLIES RUN OUT?>
<LAST NAME, FIRST NAME>
<LAB:    CLRL    R4>
^%ARGUMENT IS <LAST,FIRST> FOR CALL%
^?EXPRESSION IS <5+3>*<4+2>?
```

In the last two examples the initial circumflex indicates that the percent sign (%) and question mark (?), respectively, are the delimiters. Note that only the left hand delimiter is preceded by a circumflex.

The assembler interprets a string argument enclosed by delimiters as one actual argument and associates it with one formal argument. If a string argument that contains separator characters is not enclosed by delimiters, the assembler interprets it as successive actual arguments and associates it with successive formal arguments.

For example, the following macro call has one formal argument.

```
.MACRO  REPEAT STRNG
.ASCII  /STRNG/
.ASCII  /STRNG/
.ENDM   REPEAT
```

The following two macro calls demonstrate actual arguments with and without delimiters.

```
1.      REPEAT  <A B C D E>
        .ASCII  /A B C D E/
        .ASCII  /A B C D E/

2.      REPEAT  A B C D E
        %MACRO-E-TOOMNYARGS, Too many arguments in MACRO call
```

Note that the assembler interpreted the second macro call as having five actual arguments instead of one actual argument with spaces.

When a macro is called, the assembler removes the delimiters (if present) around a string before associating it with the formal arguments.

If a string contains a semicolon, the string must be enclosed by delimiters, or the semicolon will mark the start of the comment field.

To pass a number containing a radix or unary operator (for example, ^XF19), the entire argument must be enclosed by delimiters, or the assembler will interpret the radix operator as a delimiter. The following are macro arguments that are enclosed in delimiters because they contain radix operators:

```
<^XF19>
<^B01100011>
<^F1.5>
```

Macros can be nested, that is a macro definition can contain a call to another macro. If within a macro definition, another macro is called and passed a string argument, the programmer must delimit the argument so that the entire string is passed to the second macro as one argument.

The following macro definition contains a call to the REPEAT macro defined in an earlier example:

```
        .MACRO  CNTRPT  LAB1,LAB2,STR_ARG
LAB1:   .BYTE   LAB2-LAB1-1             ; LENGTH OF 2+STRING

        REPEAT  <STR_ARG>               ; CALL REPEAT MACRO
LAB2:
        .ENDM   CNTRPT
```

Note that the argument in the call to REPEAT is enclosed in angle brackets even though the actual argument does not contain any separator characters. This is done because the actual argument in the call to REPEAT is a formal argument in the macro definition and will be replaced with an actual argument that may contain separator characters.

The following example calls the macro CNTRPT which in turn calls the macro REPEAT:

```
        CNTRPT  ST,FIN,<LEARN YOUR ABC'S>
ST:     .BYTE   FIN-ST-1                ; LENGTH OF 2*STRING
        REPEAT  <LEARN YOUR ABC'S>      ; CALL REPEAT MACRO
        .ASCII  /LEARN YOUR ABC'S/
        .ASCII  /LEARN YOUR ABC'S/
FIN:
```

An alternative method to pass string arguments in nested macros is to enclose the macro argument in nested delimiters. In this case the macro calls in the macro definitions should not have delimiters. Each time the delimited argument is used in a macro call, the assembler removes the outermost pair of delimiters before associating it with the formal argument. This method is not recommended because it requires that the programmer know how deeply a macro is nested.

The following macro definition also contains a call to the repeat macro:

```
        .MACRO   CNTRPT2 LAB1,LAB2,STR_ARG
LAB1:   .BYTE    LAB2-LAB1-1              ; LENGTH OF 2*STRING

        REPEAT   STR_ARG                  ; CALL REPEAT MACRO

LAB2:
        .ENDM    CNTRPT2
```

Note that the argument in the call to REPEAT is not enclosed in angle brackets.

The following example calls the macro CNTRPT2:

```
        CNTRPT2 BEG,TERM,<<MIND YOUR P'S AND Q'S>>
BEG:    .BYTE    TERM-BEG-1               ; LENGTH OF 2*STRING
        REPEAT   <MIND YOUR P'S AND Q'S>  ; CALL REPEAT MACRO
        .ASCII   /MIND YOUR P'S AND Q'S/
        .ASCII   /MIND YOUR P'S AND Q'S/
TERM:
```

Note that even though the call to REPEAT in the macro definition is not enclosed in delimiters, the call in the expansion is enclosed in delimiters because the call to CNTRPT2 contains nested delimiters around the string argument.

6.1.4 Argument Concatenation

The argument concatenation operator, the apostrophe ('), concatenates a macro argument with some constant text. Apostrophes can either precede or follow a formal argument name in the macro source.

If an apostrophe precedes the argument name, the text before the apostrophe is concatenated with the actual argument when the macro is expanded. For example, if ARG1 is a formal argument associated with the actual argument TEST, ABCDE'ARG1 is expanded to ABCDETEST.

If an apostrophe follows the formal argument name, the actual argument is concatenated with the text that follows the apostrophe when the macro is expanded. For example, if ARG2 is a formal argument associated with the actual argument MOV, ARG2'L is expanded to MOVL.

Note that the apostrophe itself does not appear in the macro expansion.

To concatenate two arguments, separate the two formal arguments with two successive apostrophes. Two apostrophes are needed because each concatenation operation discards an apostrophe from the expansion.

An example of a macro definition that uses concatenation follows:

```
        .MACRO CONCAT   INST,SIZE,NUM
TEST'NUM':      INST''SIZE      R0,R'NUM
TEST'NUM'X:
        .ENDM   CONCAT
```

Note that two successive apostrophes are used when concatenating the two formal arguments INST and SIZE.

An example of a macro call and expansion follows:

```
        CONCAT  MOV,L,5
TEST5:  MOVL    R0,R5
TEST5X:
```

6.1.5 Passing Numeric Values of Symbols

When a symbol is specified as an actual argument, the name of the symbol, not the numeric value of the symbol, is passed to the macro. However, the value of the symbol can be passed by inserting a backslash before the symbol in the macro call. The assembler then passes the characters representing the decimal value of the symbol to the macro. For example, if the symbol COUNT has a value of 2 and the actual argument specified is \COUNT, the assembler passes the string "2" to the macro; it does not pass the name of the symbol, "COUNT".

Passing numeric values of symbols is especially useful with the apostrophe (') concatenation operator for creating new symbols.

An example of a macro definition for passing numeric values of symbols follows:

```
        .MACRO  TESTDEF,TESTNO,ENTRYMASK=^?^M<>?
        .ENTRY  TEST'TESTNO,ENTRYMASK    ; USES ARG CONCATENATION
        .ENDM   TESTDEF
```

The following example shows a possible call and expansion of the macro defined above:

```
COUNT = 2
        TESTDEF \COUNT
        .ENTRY  TEST2,^M<>
COUNT = COUNT + 1
        TESTDEF \COUNT,^?^M<R3,R4>?
        .ENTRY  TEST3,^M<R3,R4>
```

6.1.6 Created Local Labels

Local labels are often very useful in macros. Although the programmer can specify local labels in the macro definition, these local labels might be duplicated elsewhere in the local label block and might thus cause errors. However, the programmer can use the assembler to create local labels in the macro expansion which will not conflict with other local labels. These labels are called created local labels.

Created local labels range from 30000$ through 65535$. Each time the assembler creates a new local label, it increments the numeric part of the label name by 1. Consequently, no user-defined local labels should be in the range of 30000$ through 65535$.

The programmer specifies a created local label by a question mark (?) placed in front of the formal argument name. When the macro is expanded, the assembler creates a new local label if the corresponding actual argument is blank. If the corresponding actual argument is specified, the assembler substitutes the actual argument for the formal argument. Created local symbols can be used only in the first 31 formal arguments specified in the .MACRO directive.

Created local labels can be associated only with positional actual arguments; created local labels cannot be associated with keyword actual arguments.

The following example is a macro definition specifying a created local label:

```
        .MACRO  POSITIVE        ARG1,?L1
        TSTL    ARG1
        BGEQ    L1
        MNEGL   ARG1,ARG1
L1:     .ENDM   POSITIVE
```

The following three calls and expansions of the macro defined above show both created local labels and a user-specified local label:

```
1.      POSITIVE        R0
        TSTL    R0
        BGEQ    30000$
        MNEGL   R0,R0
30000$:

2.      POSITIVE        COUNT
        TSTL    COUNT
        BGEQ    30001$
        MNEGL   COUNT,COUNT
30001$:

3.      POSITIVE        VALUE,10$
        TSTL    VALUE
        BGEQ    10$
        MNEGL   VALUE,VALUE
10$:
```

6.1.7 Macro String Operators

The three macro string operators are:

- %LENGTH
- %LOCATE
- %EXTRACT

These operators perform string manipulations on macro arguments and ASCII strings. They can be used only in macros and repeat blocks. The following sections describe these operators and give their formats and examples of their use.

%LENGTH

6.1.7.1 **%LENGTH Operator** - The %LENGTH operator returns the length of a string. For example, the value of %LENGTH(<ABCDE>) is 5.

Format

%LENGTH(string)

Parameters

string

A macro argument or a delimited string. The string can be delimited by angle brackets or a character preceded by a circumflex (see Section 6.1.3).

Examples

Macro definition:

```
.MACRO  CHK_SIZE    ARG1               ; MACRO CHECKS IF ARG1
.IF GREATER_EQUAL   %LENGTH(ARG1)-3    ; IS BETWEEN 3 AND
.IF LESS_THAN       6-%LENGTH(ARG1)    ; 6 CHARACTERS LONG
.ERROR          ; ARGUMENT ARG1 IS GREATER THAN 6 CHARACTERS
.ENDC                                  ; IF MORE THAN 6
.IF_FALSE                              ; IF LESS THAN 3
.ERROR          ; ARGUMENT ARG1 IS LESS THAN 3 CHARACTERS
.ENDC                                  ; OTHERWISE DO
.ENDM   CHK_SIZE                       ; NOTHING
```

Macro calls and expansions of the macro defined above:

```
1.      CHK_SIZE        A                      ; SHOULD BE TOO SHORT
        .IF GREATER_EQUAL   1-3 ; IS BETWEEN 3 AND
        .IF LESS_THAN       6-1 ; 6 CHARACTERS LONG
        .ERROR          ; ARGUMENT A IS GREATER THAN 6 CHARACTERS
        .ENDC                                  ; IF MORE THAN 6
        .IF_FALSE                              ; IF LESS THAN 3
%MACRO-E-GENERR, Generated ERROR: ARGUMENT A IS LESS THAN 3 CHARACTERS

        .ENDC                                  ; OTHERWISE DO

2.      CHK_SIZE        ABC                    ; SHOULD BE OK
        .IF GREATER_EQUAL   3-3 ; IS BETWEEN 3 AND
        .IF LESS_THAN       6-3 ; 6 CHARACTERS LONG
        .ERROR          ; ARGUMENT ABC IS GREATER THAN 6 CHARACTERS
        .ENDC                                  ; IF MORE THAN 6
        .IF_FALSE                              ; IF LESS THAN 3
        .ERROR          ; ARGUMENT ABC IS LESS THAN 3 CHARACTERS
        .ENDC                                  ; OTHERWISE DO
```

%LOCATE

6.1.7.2 **%LOCATE Operator** - The %LOCATE operator locates a substring within a string. If %LOCATE finds a match of the substring, it returns the character position of the first character of the match in the string. For example, the value of %LOCATE(<D>,<ABCDEF>) is 3. Note that the first character position of a string is 0. If %LOCATE does not find a match, it returns a value equal to the length of the string. For example, the value of %LOCATE(<Z>,<ABCDEF>) is 6.

The %LOCATE operator returns a numeric value that can be used in any expression.

Format

```
%LOCATE(string1,string2 [,symbol] )
```

Parameters

string1

A string that specifies the substring. The substring can be either a macro argument or a delimited string. The delimiters can be either angle brackets or a character preceded by a circumflex.

string2

The string that is searched for the substring. The string can be either a macro argument or a delimited string. The delimiters can be either angle brackets or a character preceded by a circumflex.

symbol

An optional symbol or decimal number that specifies the position in string2 at which the assembler should start the search. If this argument is omitted, the assembler starts the search at position 0 (the beginning of the string). A symbol must be an absolute symbol that has been previously defined and a number must be an unsigned decimal number. Expressions and radix operators are not allowed.

Example

Macro definition:

```
.MACRO  BIT_NAME    ARG1                    ; CHECKS IF ARG1 IS IN LIST
.IF EQUAL       %LOCATE(ARG1,<DELDFWDLTDMOESC>)-15
                                            ; IF IT IS NOT PRINT ERROR
.ERROR          ; ARG1 IS AN INVALID BIT NAME
.ENDC                                       ; IF IT IS DO
.ENDM   BIT_NAME                            ; NOTHING
```

Macro calls and expansions of the macro defined above:

```
1.      BIT_NAME        ESC                     ; IS IN LIST
        .IF_EQUAL       12-15
                                                ; IF IT IS NOT PRINT ERROR
        .ERROR          ; ESC IS AN INVALID BIT NAME
        .ENDC                                   ; IF IT IS DO

2.      BIT_NAME        FOO                     ; NOT IN LIST
        .IF_EQUAL       15-15
                                                ; IF IT IS NOT PRINT ERROR
%MACRO-E-GENERR, Generated ERROR:  FOO IS AN INVALID BIT NAME

        .ENDC                                   ; IF IT IS DO
```

Note

If the optional symbol is specified, the search begins at the character position of string2 specified by the symbol. For example, the value of %LOCATE(<ACE>,<SPACE_HOLDER>,5) is 12 because there is no match after the 5th character position.

%EXTRACT

6.1.7.3 **%EXTRACT Operator** - The %EXTRACT operator extracts a substring from a string. It returns the substring that begins at the specified position and is the specified length. For example, the value of %EXTRACT(2,3,<ABCDEF>) is CDE. Note that the first character in a string is in position 0.

Format

%EXTRACT(symbol1,symbol2,string)

Parameters

symbol1

A symbol or decimal number that specifies the starting position of the substring. A symbol must be an absolute symbol that has been previously defined and a number must be an unsigned decimal number. Expressions and radix operators are not allowed.

symbol2

A symbol or decimal number that specifies the length of the substring. A symbol must be an absolute symbol that has been previously defined and a number must be an unsigned decimal number. Expressions and radix operators are not allowed.

string

A macro argument or a delimited string. The string can be delimited by angle brackets or a character preceded by a circumflex.

Example

Macro definition:

```
        .MACRO  RESERVE ARG1
XX = %LOCATE(<=>,ARG1)
        .IF EQUAL   XX-%LENGTH(ARG1)
        .WARN   ; INCORRECT FORMAT FOR MACRO CALL - ARG1
        .MEXIT
        .ENDC

%EXTRACT(0,XX,ARG1)::
XX = XX+1
        .BLKB   %EXTRACT(XX,3,ARG1)
        .ENDM   RESERVE
```

Macro calls and expansions of the macro defined above:

```
1.      RESERVE FOOBAR
XX = 6
        .IF EQUAL  XX-6
%MACRO-W-GENWRN, Generated WARNING:  INCORRECT FORMAT FOR MACRO CALL - FOOBA

        .MEXIT

2.      RESERVE LOCATION=12
XX = 8
        .IF EQUAL  XX-11
        .WARN   ; INCORRECT FORMAT FOR MACRO CALL - LOCATION=12
        .MEXIT
        .ENDC

LOCATION::
XX = XX+1
        .BLKB   12
```

Notes

If the starting position specified is greater than or equal to the length of the string, %EXTRACT returns a null string (a string of 0 characters). If the length specified is 0, %EXTRACT returns a null string.

6.2 MACRO DIRECTIVES

The remainder of this chapter describes the macro directives in detail, showing their formats and giving examples of their use. The directives are presented in alphabetical order.

.ENDM

.ENDM--END DEFINITION DIRECTIVE

.ENDM terminates the macro definition. See the description of .MACRO for an example of the use of .ENDM.

Format

```
.ENDM [macro-name]
```

Parameter

macro-name

The name of the macro whose definition is to be terminated. The macro name is optional; but, if specified, it must match the name defined in the matching .MACRO directive. The macro name should be specified so that the assembler can detect any improperly nested macro definitions.

Note

If .ENDM is encountered outside a macro definition, the assembler displays an error message.

.ENDR

.ENDR--END RANGE DIRECTIVE

.ENDR indicates the end of a repeat range. It must be the final statement of every indefinite repeat block directive (.IRP and .IRPC) and every repeat block directive (.REPEAT). See the description of these directives for examples of the use of .ENDR.

Format

```
.ENDR
```

.IRP

.IRP--INDEFINITE REPEAT ARGUMENT DIRECTIVE

.IRP replaces a formal argument with successive actual arguments specified in an argument list. This replacement process occurs during the expansion of the indefinite repeat block range. The .ENDR directive specifies the end of the range.

.IRP is analogous to a macro definition with only one formal argument. At each expansion of the repeat block, this formal argument is replaced with successive elements from the argument list. The directive and its range are coded inline within the source program. This type of macro definition and its range do not require calling the macro by name, as do other macros described in this chapter.

.IRP can appear either within or outside another macro definition, indefinite repeat block, or repeat block (see the description of .REPEAT). The rules for specifying .IRP arguments are the same as those for specifying macro arguments.

Format

```
.IRP symbol,<argument list>
.
.
.
range
.
.
.
.ENDR
```

Parameters

symbol

A formal argument that is successively replaced with the specified actual arguments enclosed in angle brackets. If no formal argument is specified, the assembler displays an error message.

<argument list>

A list of actual arguments enclosed in angle brackets and used in expanding the indefinite repeat range. An actual argument can consist of one or more characters; multiple arguments must be separated by a legal separator (comma, space, or tab). If no actual arguments are specified, no action is taken.

range

The block of source text to be repeated once for each occurrence of an actual argument in the list. The range can contain macro definitions and repeat ranges. .MEXIT is legal within the range.

Example

Macro definition:

```
.MACRO   CALL_SUB          SUBR,A1,A2,A3,A4,A5,A6,A7,A8,A9,A10
.NARG    COUNT
.IRP     ARG,<A10,A9,A8,A7,A6,A5,A4,A3,A2,A1>
.IIF NOT_BLANK             ARG,     PUSHL ARG
.ENDR
CALLS    #<COUNT-1>,SUBR            ; NOTE SUBR IS COUNTED
.ENDM    CALL_SUB
```

Macro call and expansion of the macro defined above:

```
CALL_SUB         TEST,INRES,INTES,UNLIS,OUTCON,#205
.NARG    COUNT
.IRP     ARG,<,,,,,#205,OUTCON,UNLIS,INTES,INRES>
.IIF NOT_BLANK   ARG,     PUSHL   ARG
.ENDR
.IIF NOT_BLANK   ,        PUSHL
.IIF NOT_BLANK   ,        PUSHL
.IIF NOT_BLANK   ,        PUSHL
.IIF NOT_BLANK   ,        PUSHL
.IIF NOT_BLANK   ,        PUSHL
.IIF NOT_BLANK   #205,    PUSHL   #205
.IIF NOT_BLANK   OUTCON,  PUSHL   OUTCON
.IIF NOT_BLANK   UNLIS,   PUSHL   UNLIS
.IIF NOT_BLANK   INTES,   PUSHL   INTES
.IIF NOT_BLANK   INRES,   PUSHL   INRES
CALLS    #<COUNT-1>,TEST            ; NOTE TEST IS COUNTED
```

This example uses the .NARG directive to count the arguments and the .IIF NOT_BLANK directive (see descriptions of .IF and .IIF in Chapter 5). to determine whether the actual argument is blank. If the argument is blank, no binary code is generated.

.IRPC

.IRPC--INDEFINITE REPEAT CHARACTER DIRECTIVE

.IRPC is similar to .IRP except that .IRPC permits single-character substitution, rather than argument substitution. On each iteration of the indefinite repeat range, the formal argument is replaced with each successive character in the specified string. The .ENDR directive specifies the end of the range.

.IRPC is analogous to a macro definition with only one formal argument. At each expansion of the repeat block, this formal argument is replaced with successive characters from the actual argument string. The directive and its range are coded inline within the source program and do not require calling the macro by name, as do other macros described in this chapter.

.IRPC can appear either within or outside another macro definition, indefinite repeat block, or repeat block (see description of .REPEAT).

Format

```
.IRPC   symbol,<string>
.
.
.
range
.
.
.
.ENDR
```

Parameters

symbol

A formal argument that is successively replaced with the specified characters enclosed in angle brackets. If no formal argument is specified, the assembler displays an error message.

<string>

A sequence of characters enclosed in angle brackets and used in the expansion of the indefinite repeat range. Although the angle brackets are required only when the string contains separating characters, their use is recommended for legibility.

range

The block of source text to be repeated once for each occurrence of a character in the list. The range can contain macro definitions and repeat ranges. .MEXIT is legal within the range.

Example

Macro Definition:

```
        .MACRO  HASH_SYM        SYMBOL
        .NCHR   HV,<SYMBOL>
        .IRPC   CHR,<SYMBOL>
HV = HV+^A?CHR?
        .ENDR
        .ENDM   HASH_SYM
```

Macro call and expansion of the macro defined above:

```
        HASH_SYM        <MOVC5>
        .NCHR   HV,<MOVC5>
        .IRPC   CHR,<MOVC5>
HV = HV+^A?CHR?
        .ENDR
HV = HV+^A?M?
HV = HV+^A?O?
HV = HV+^A?V?
HV = HV+^A?C?
HV = HV+^A?5?
```

This example uses the .NCHR directive to count the number of characters in actual argument.

.LIBRARY

.LIBRARY--MACRO LIBRARY DIRECTIVE

.LIBRARY adds a name to the macro library list that is searched whenever a .MCALL or an undefined opcode is encountered. The libraries are searched in the reverse order in which they were specified to the assembler.

If the programmer omits any information from the macro-library-name argument, default values are assumed. The device defaults to the user's disk; the directory defaults to the user's directory; and the file type defaults to MLB.

DIGITAL recommends that libraries be specified in the MACRO command line with the /LIBRARY qualifier rather than with the .LIBRARY directive. The .LIBRARY directive makes moving files cumbersome.

Format

```
.LIBRARY macro-library-name
```

Parameter

macro-library-name

A delimited string that is the file specification of a macro library.

Example

```
.LIBRARY  /DB1:[TEST]USERM/     ; MACRO LIBRARY USERM.MLB
.LIBRARY  ?DB1:SYSDEF.MLB?
.LIBRARY  \CURRENT.MLB\
```

.MACRO

.MACRO--MACRO DEFINITION DIRECTIVE

.MACRO begins the definition of a macro. It gives the macro name and a list of formal arguments (see Section 6.1). If the name specified is the same as the name of a previously defined macro, the previous definition is deleted and replaced with the new one. The .MACRO directive is followed by the source text to be included in the macro expansion. The .ENDM directive specifies the end of the range.

Macro names do not conflict with user-defined symbols. A macro and a user-defined symbol can both have the same name.

When the assembler encounters a .MACRO directive, it adds the macro name to its macro name table and stores the source text of the macro (up to the matching .ENDM directive). No other processing occurs until the macro is expanded.

The symbols in the formal argument list are associated with the macro name and are limited to the scope of the definition of that macro. For this reason, the symbols that appear in the formal argument list can also appear elsewhere in the program.

Format

```
.MACRO macro-name  [formal-argument-list]
  .
  .
  .
range
  .
  .
  .
.ENDM [macro name]
```

Parameters

macro-name

The name of the macro to be defined; this name can be any legal symbol up to 31 characters long.

formal-argument-list

The symbols, separated by commas, to be replaced by the actual arguments in the macro call.

range

The source text to be included in the macro expansion.

Example

Macro definition:

```
        .MACRO  USERDEF
        .PSECT  DEFS,ABS
MYSYM=  5
HIVAL=  ^XFFF123
LOWVAL= 0
        .PSECT  RWDATA,NOEXE,LONG
TABLE:  .BLKL   100
LIST:   .BLKB   10
        .MACRO  USERDEF                 ; REDEFINE IT TO NULL
        .ENDM   USERDEF
        .ENDM   USERDEF
```

Macro calls and expansions of the macro defined above:

```
1.      USERDF          ; SHOULD EXPAND DATA
        .PSECT  DEFS,ABS
MYSYM=  5
HIVAL=  ^XFFF123
LOWVAL= 0
        .PSECT  RWDATA,NOEXE,LONG
TABLE:  .BLKL   100
LIST:   .BLKB   10
        .MACRO  USERDEF                 ; REDEFINE IT TO NULL
        .ENDM   USERDEF

2.      USERDF          ; SHOULD EXPAND NOTHING
```

In this example, when the macro is called the first time it defines some symbols and data storage areas and then redefines itself. When the macro is called a second time, the macro expansion contains no source text.

Notes

1. If a macro has the same name as a VAX-11 opcode, the macro is used instead of the instruction. This feature allows a programmer to temporarily redefine an opcode.

2. If a macro has the same name as a VAX-11 opcode and is in a macro library, the .MCALL directive must be used to define the macro. Otherwise, because the symbol is already defined (as the opcode), the assembler will not search the macro libraries.

3. The programmer can redefine a macro with new source text during assembly by specifying a second .MACRO directive with the same name. Including a second .MACRO directive within the original macro definition causes the first macro call to redefine the macro. This is useful when a macro performs initialization or defines symbols; that is, when an operation is performed only once. The macro redefinition can eliminate unneeded source text in a macro or it can delete the entire macro. The .MDELETE directive provides another way to delete macros.

.MCALL

.MCALL--MACRO CALL DIRECTIVE

.MCALL specifies the names of the system and/or user-defined macros that are required to assemble the source program but are not defined in the source file.

If any named macro is not found upon completion of the search (that is, if the macro is not defined in any of the macro libraries), the assembler displays an error message.

Format

.MCALL macro-name-list

Parameter

macro-name-list

A list of macros to be defined for this assembly. The names must be separated by commas.

Example

```
.MCALL   INSQUE          ; SUBSTITUTE MACRO IN
                         ; LIBRARY FOR INSQUE
                         ; INSTRUCTION
```

Note

.MCALL is provided for compatibility with MACRO-11; DIGITAL recommends that it not be used. When VAX-11 MACRO finds an unknown symbol in the opcode field, it automatically searches all macro libraries. If it finds the symbol in a library, it uses the macro definition and expands the macro reference. If VAX-11 MACRO does not find the unknown symbol in the library, it displays an error message. There is one exception for which .MCALL must be used: when a macro has the same name as an opcode (see description of .MACRO).

.MDELETE

.MDELETE--MACRO DELETION DIRECTIVE

.MDELETE deletes the definitions of specified macros. The number of macros actually deleted is printed in the assembly listing on the same line as the .MDELETE directive.

.MDELETE completely deletes the macro, freeing memory as necessary, whereas the technique of macro redefinition explained in the description of .MACRO merely redefines the macro.

Format

```
.MDELETE macro-name-list
```

Parameter

macro-name-list

A list of macros whose definitions are to be deleted. The names must be separated by commas.

Example

```
.MDELETE     USERDEF,$SSDEF,ALTR
```

.MEXIT

.MEXIT--MACRO EXIT DIRECTIVE

.MEXIT terminates a macro expansion before the end of the macro. Termination is the same as if .ENDM was encountered. The directive can also be used within repeat blocks. .MEXIT is most useful in conditional expansion of macros because it bypasses the complexities of nested conditional directives and alternate assembly paths.

Format

```
.MEXIT
```

Example

```
.MACRO  POLO    N,A,B
  .
  .
.IF EQ  N                       ; START CONDITIONAL ASSEMBLY BLOCK.
  .
  .
.MEXIT                          ; TERMINATE MACRO EXPANSION.
.ENDC                           ; END CONDITIONAL ASSEMBLY BLOCK.
  .
  .
.ENDM   POLO                    ; NORMAL END OF MACRO.
```

In this example, if the actual argument for the formal argument N equals 0, the conditional block would be assembled, and the macro expansion would be terminated by .MEXIT.

Notes

1. When .MEXIT occurs in a repeat block, the assembler terminates the current repetition of the range and suppresses further expansion of the repeat range.

2. When macros or repeat blocks are nested, .MEXIT exits to the next higher level of expansion.

3. If .MEXIT occurs outside a macro definition or a repeat block, the assembler displays an error message.

.NARG

.NARG--NUMBER OF ARGUMENTS DIRECTIVE

.NARG determines the number of arguments in the current macro call.

.NARG counts all the positional arguments specified in the macro call, including null arguments (specified by adjacent commas). The value assigned to the specified symbol does not include either any keyword arguments or any formal arguments that have default values.

Format

```
.NARG    symbol
```

Parameter

symbol

A symbol that is assigned a value equal to the number of arguments in the macro call.

Example

Macro definition:

```
        .MACRO  CNT_ARG A1,A2,A3,A4,A5,A6,A7,A8,A9=DEF9,A10=DEF10
        .NARG   COUNTER                 ; COUNTER IS SET TO NO. OF ARGS
        .WORD   COUNTER                 ; STORE VALUE OF COUNTER
        .ENDM   CNT_ARG
```

Macro calls and expansions of the macro defined above:

```
1.      CNT_ARG TEST,FIND,ANS           ; COUNTER WILL = 3
        .NARG   COUNTER                 ; COUNTER IS SET TO NO. OF ARGS
        .WORD   COUNTER                 ; STORE VALUE OF COUNTER

2.      CNT_ARG                         ; COUNTER WILL = 0
        .NARG   COUNTER                 ; COUNTER IS SET TO NO. OF ARGS
        .WORD   COUNTER                 ; STORE VALUE OF COUNTER

3.      CNT_ARG TEST,A2=SYMB2,A3=SY3    ; COUNTER WILL = 1
        .NARG   COUNTER                 ; COUNTER IS SET TO NO. OF ARGS
        .WORD   COUNTER                 ; STORE VALUE OF COUNTER
                                        ; KEYWORD ARGUMENTS ARE NOT COUNTE

4.      CNT_ARG ,SYMBL,,                ; COUNTER WILL = 3
        .NARG   COUNTER                 ; COUNTER IS SET TO NO. OF ARGS
        .WORD   COUNTER                 ; STORE VALUE OF COUNTER
                                        ; NULL ARGUMENTS ARE COUNTED
```

Note

If .NARG appears outside of a macro, the assembler displays an message.

.NCHR

.NCHR--NUMBER OF CHARACTERS DIRECTIVE

.NCHR determines the number of characters in a specified character string. It can appear anywhere in a VAX-11 MACRO program and is useful in calculating the length of macro arguments.

Format

```
.NCHR     symbol,<string>
```

Parameters

symbol

A symbol that is assigned a value equal to the number of characters in the specified character string.

<string>

A sequence of printable characters. The character string must be delimited by angle brackets or a character preceded by a circumflex only if the specified character string contains a legal separator (comma, space, and/or tab) or a semicolon.

Example

Macro definition:

```
.MACRO  CHAR    MESS             ; DEFINE MACRO
.NCHR   CHRCNT,<MESS>            ; ASSIGN VALUE TO CHRCNT
.WORD   CHRCNT                   ; STORE VALUE
.ASCII  /MESS/                   ; STORE CHARACTERS
.ENDM   CHAR                     ; FINISH
```

Macro calls and expansions of the macro defined above:

```
1.      CHAR     <HELLO>                 ; CHRCNT WILL = 5
        .NCHR    CHRCNT,<HELLO>          ; ASSIGN VALUE TO CHRCNT
        .WORD    CHRCNT                  ; STORE VALUE
        .ASCII   /HELLO/                 ; STORE CHARACTERS

2.      CHAR     <14, 75.39  4>          ; CHRCNT WILL = 12(DEC)
        .NCHR    CHRCNT,<14, 75.39  4>   ; ASSIGN VALUE TO CHRCNT
        .WORD    CHRCNT                  ; STORE VALUE
        .ASCII   /14, 75.39  4/          ; STORE CHARACTERS
```

.NTYPE

.NTYPE--OPERAND TYPE DIRECTIVE

.NTYPE determines the addressing mode of the specified operand.

The value of the symbol is set to the specified addressing mode. In most cases, an 8-bit (1-byte) value is returned. Bits 0 through 3 specify the register associated with the mode, and bits 4 through 7 specify the addressing mode. To provide concise addressing information, the mode bits 4 through 7 are not exactly the same as the numeric value of the addressing mode described in Table 4-1. Specifically, literal mode is indicated by a 0 in bits 4 through 7 instead of the values 0 through 3 described in Table 4-1. Mode 1 indicates an immediate mode operand, mode 2 indicates an absolute mode operand, and mode 3 indicates a general mode operand.

For indexed addressing mode, a 16-bit (2-byte) value is returned. The high-order byte contains the addressing mode of the base operand specifier and the low-order byte contains the addressing mode of the primary operand (the index register).

See the VAX-11 Architecture Handbook and Chapter 4 of this manual for more information on addressing modes.

Format

```
.NTYPE     symbol,operand
```

Parameter

symbol

Any legal VAX-11 MACRO symbol. This symbol is assigned a value equal to the 8- or 16-bit addressing mode of the operand argument that follows.

operand

Any legal address expression, as used with an opcode. If no argument is specified, 0 is assumed.

Example

Macro Definition:

```
;
; THE FOLLOWING MACRO IS USED TO PUSH AN ADDRESS ON THE STACK.  IT CHECKS
; THE OPERAND TYPE (BY USING .NTYPE) TO DETERMINE IF THE OPERAND IS AN
; ADDRESS AND, IF NOT, THE MACRO SIMPLY PUSHES THE ARGUMENT ON THE STACK
; AND GENERATES A WARNING MESSAGE.
;
        .MACRO  PUSHADR ADDR
        .NTYPE  A,ADDR                  ; ASSIGNS OPERAND TYPE TO A
A = A@-4&^XF                            ; ISOLATE ADDRESSING MODE
        .IF IDENTICAL 0,<ADDR>          ; IS ARGUMENT EXACTLY 0
        PUSHL   #0                      ; STACK ZERO
        .MEXIT                          ; EXIT FROM MACRO
        .ENDC
ERR = 0                                 ; ERR TELLS IF MODE IS ADDRESS
                                        ; ERR = 0 FOR ADDRESS, 1 WHEN NOT
        .IIF LESS_EQUAL A-1,    ERR=1   ; IS MODE NOT LITERAL OR IMMEDIATE
        .IIF EQUAL A-5,    ERR=1        ; IS MODE NOT REGISTER
        .IF EQUAL  ERR                  ; IS MODE ADDRESS?
        PUSHAL  ADDR                    ; YES, STACK ADDRESS
        .IFF                            ; NO
        PUSHL   ADDR                    ; THEN STACK OPERAND & WARN
        .WARN   ; ADDR IS NOT AN ADDRESS
        .ENDC
        .ENDM   PUSHADR
```

Macro calls and expansions of the macro defined above:

```
1.      PUSHADR (R0)                    ; VALID ARGUMENT
        PUSHAL  (R0)                    ; YES, STACK ADDRESS

2.      PUSHADR (R1)[R4]                ; VALID ARGUMENT
        PUSHAL  (R1)[R4]                ; YES, STACK ADDRESS

3.      PUSHADR 0                       ; IS ZERO
        PUSHL   #0                      ; STACK ZERO

4.      PUSHADR #1                      ; NOT AN ADDRESS
        PUSHL   #1                      ; THEN STACK OPERAND & WARN
%MACRO-W-GENWRN, Generated WARNING: #1 IS NOT AN ADDRESS

5.      PUSHADR R0                      ; NOT AN ADDRESS
        PUSHL   R0                      ; THEN STACK OPERAND & WARN
%MACRO-W-GENWRN, Generated WARNING: R0 IS NOT AN ADDRESS
```

Note that to save space, this example is listed as it would appear if .SHOW BINARY, not .SHOW EXPANSIONS, was specified in the source program.

.REPEAT

.REPEAT--REPEAT BLOCK DIRECTIVE

.REPEAT repeats a block of code, a specified number of times, inline with other source code. The .ENDR directive specifies the end of the range.

Format

```
.REPEAT expression
.
.
.
range
.
.
.
.ENDR
```

Parameters

expression

An expression whose value controls the number of times the range is to be assembled within the program. When the expression is less than or equal to 0, the repeat block is not assembled. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5).

range

The source text to be repeated the number of times specified by the value of the expression. The repeat block can contain macro definitions, indefinite repeat blocks, or other repeat blocks. .MEXIT is legal within the range.

Example

Macro definition:

```
.MACRO  COPIES  STRING,NUM
.REPEAT NUM
.ASCII  /STRING/
.ENDR
.BYTE   0
.ENDM   COPIES
```

Macro calls and expansions of the macro defined above:

```
1.      COPIES  <ABCDEF>,5
        .REPEAT 5
        .ASCII  /ABCDEF/
        .ENDR
        .ASCII  /ABCDEF/
        .ASCII  /ABCDEF/
        .ASCII  /ABCDEF/
        .ASCII  /ABCDEF/
        .ASCII  /ABCDEF/
        .BYTE   0
```

2.

```
VARB = 3
        COPIES  <HOW MANY TIMES>,\VARB
        .REPEAT 3
        .ASCII  /HOW MANY TIMES/
        .ENDR
        .ASCII  /HOW MANY TIMES/
        .ASCII  /HOW MANY TIMES/
        .ASCII  /HOW MANY TIMES/
        .BYTE   0
```

Note

The alternate form of .REPEAT is .REPT.

APPENDIX A

ASCII CHARACTER SET

Table A-1 lists the ASCII characters and the hexadecimal code for each.

Table A-1
Hexadecimal/ASCII Conversion

HEX Code	ASCII Char.	HEX Code	ASCII Char.	HEX Code	ASCII Char.	HEX Code	ASCII Char.
00	NUL	20	SP	40	@	60	\
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	
1C	FS	3C	<	5C	\	7C	
1D	GS	3D	=	5D	]	7D	
1E	RS	3E	>	5E	^	7E	
1F	US	3F	?	5F		7F	DEL

APPENDIX B

VAX-11 MACRO ASSEMBLER DIRECTIVES AND LANGUAGE SUMMARY

B.1 ASSEMBLER DIRECTIVES

The following table summarizes the VAX-11 MACRO assembler directives.

Table B-1
Assembler Directives

Format	Operation
.ADDRESS address-list	Stores successive longwords of address data
.ALIGN keyword [,expression]	Aligns the location counter to the boundary specified by the keyword
.ALIGN integer [,expression]	Aligns location counter to the boundary specified by (2^integer)
.ASCIC string	Stores the ASCII string string (enclosed in delimiters), preceded by a count byte
.ASCID string	Stores the ASCII (enclosed in delimiters), preceded by a string descriptor
.ASCII string	Stores the ASCII string (enclosed in delimiters)
.ASCIZ string	Stores the ASCII string (enclosed in delimiters) followed by a 0 byte.
.BLKA expression	Reserves longwords of address data
.BLKB expression	Reserves bytes for data
.BLKD expression	Reserves quadwords for double-precision, floating-point data

(continued on next page)

Table B-1 (Cont.)
Assembler Directives

Format	Operation
.BLKF expression	Reserves longwords for single-precision, floating-point data
.BLKG expression	Reserves quadwords for floating-point data
.BLKH expression	Reserves octawords for extended-precision floating-point data
.BLKL expression	Reserves longwords for data
.BLKO expression	Reserves octawords for data
.BLKQ expression	Reserves quadwords for data
.BLKW expression	Reserves words for data
.BYTE expression-list	Generates successive bytes of data; each byte contains the value of the specified expression
.CROSS	Enables cross-referencing of all symbols
.CROSS symbol-list	Cross-references specified symbols
.DEBUG symbol-list	Makes symbol names known to the debugger
.DEFAULT DISPLACEMENT, keyword	Specifies the default displacement length for the relative addressing modes
.D_FLOATING literal-list	Generates 8-byte, double-precision, floating-point data
.DISABLE argument-list	Disables function(s) specified in argument-list
.DOUBLE literal-list	Equivalent to .D_FLOATING
.DSABL argument-list	Equivalent to .DISABLE
.ENABL argument-list	Equivalent to .ENABLE
.ENABLE argument-list	Enables function(s) specified in argument-list
.END [symbol]	Indicates logical end of source program; optional symbol specifies transfer address

(continued on next page)

Table B-1 (Cont.)
Assembler Directives

Format	Operation
.ENDC	Indicates end of conditional assembly block
.ENDM [macro-name]	Indicates end of macro definition
.ENDR	Indicates end of repeat block
.ENTRY symbol [,expression]	Procedure entry directive
.ERROR [expression] ;comment	Displays specified error message
.EVEN	Ensures that the current location counter has an even value (adds 1 if it is odd)
.EXTERNAL symbol-list	Indicates specified symbols are externally defined
.EXTRN symbol-list	Equivalent to .EXTERNAL
.F_FLOATING literal-list	Generates 4-byte, single-precision, floating point data
.FLOAT literal-list	Equivalent to .F_FLOATING
.G_FLOATING literal-list	Generates 8-byte G_floating-point data
.GLOBAL symbol-list	Indicates specified symbols are global symbols
.GLOBL	Equivalent to .GLOBAL
.H_FLOATING literal-list	Generates 16-byte, extended precision H_floating-point data
.IDENT string	Provides means of labeling object module with additional data
.IF condition argument(s)	Begins a conditional assembly block of source code which is included in the assembly only if the stated condition is met with respect to the argument(s) specified
.IFF	Equivalent to .IF_FALSE
.IF_FALSE	Appears only within a conditional assembly block; begins block of code to be assembled if the original condition tests false
.IFT	Equivalent to .IF_TRUE

(continued on next page)

Table B-1 (Cont.)
Assembler Directives

Format	Operation
.IFTF	Equivalent to .IF_TRUE_FALSE
.IF_TRUE	Appears only within a conditional assembly block; begins block of code to be assembled if the original condition tests true
.IF_TRUE_FALSE	Appears only within a conditional assembly block; begins block of code to be assembled unconditionally
.IIF condition argument(s), statement	Acts as a 1-line conditional assembly block where the condition is tested for the argument specified; the statement is assembled only if the condition tests true
.IRP sym<argument list>	Replaces a formal argument with successive actual arguments specified in an argument list
.IRPC sym,<string>	Replaces a formal argument with successive single characters specified in string
.LIBRARY macro-library-name	Specifies a macro library
.LIST [argument-list]	Equivalent to .SHOW
.LONG expression-list	Generates successive longwords of data; each longword contains the value of the specified expression.
.MACRO macro-name ,argument-list	Begins a macro definition
.MASK symbol [,expression]	Reserves a word for and copies a register save mask
.MCALL macro-name-list	Specifies the system and/or user-defined macros in libraries that are required to assemble the source program
.MDELETE macro-name-list	Deletes from memory the macro definitions of the macros in the list
.MEXIT	Exits from the expansion of a macro before the end of the macro is encountered

(continued on next page)

Table B-1 (Cont.)
Assembler Directives

Format	Operation
.NARG symbol	Determines the number of arguments in the current macro call
.NCHR symbol,<string>	Determines the number of characters in a specified character string
.NLIST [argument-list]	Equivalent to .NOSHOW
.NOCROSS	Disables cross-referencing of all symbols
.NOCROSS symbol-list	Disables cross-referencing of specified symbols
.NOSHOW	Decrements listing level count
.NOSHOW argument-list	Controls listing of macros and conditional assembly blocks
.NTYPE symbol,operand	Can appear only within a macro definition; equates the symbol to the addressing mode of the specified operand
.OCTA literal	Stores 16 bytes of data
.OCTA symbol	Stores 16 bytes of data
.ODD	Ensures that the current location counter has an odd value (adds 1 if it is even)
.OPDEF opcode value, operand-descriptor-list	Defines an opcode and its operand list
.PACKED decimal-string [,symbol]	Generates packed decimal data, 2 digits per byte
.PAGE	Causes the assembly listing to skip to the top of the next page, and to increment the page count
.PRINT [expression] ;comment	Displays the specified message
.PSECT	Begins or resumes the blank program section
.PSECT section-name argument-list	Begins or resumes a user-defined program section

(continued on next page)

Table B-1 (Cont.)
Assembler Directives

Format	Operation
.QUAD literal	Stores 8 bytes of data
.QUAD symbol	Stores 8 bytes of data
.REF1 operand	Generates byte operand
.REF2 operand	Generates word operand
.REF4 operand	Generates longword operand
.REF8 operand	Generates quadword operand
.REF16 operand	Generates octaword operand
.REPEAT expression	Begins a repeat block; the section of code up to the next .ENDR directive is repeated the number of times specified by the expression
.REPT	Equivalent to .REPEAT
.RESTORE	Equivalent to .RESTORE_PSECT
.RESTORE_PSECT	Restores program section context from the program section context stack
.SAVE [LOCAL_BLOCK]	Equivalent to .SAVE_PSECT
.SAVE_PSECT [LOCAL_BLOCK]	Saves current program section context on the program section context stack
.SBTTL comment-string	Equivalent to .SUBTITLE
.SHOW	Increments listing level count
.SHOW argument-list	Controls listing of macros and conditional assembly blocks
.SIGNED_BYTE expression-list	Stores successive bytes (8 bits) of signed data
.SIGNED_WORD expression-list	Stores successive words (16 bits) of signed data
.SUBTITLE comment-string	Causes the specified string to be printed as part of the assembly listing page header; the string component of each .SUBTITLE is collected into a table of contents at the beginning of the assembly listing

(continued on next page)

Table B-1 (Cont.)
Assembler Directives

Format	Operation
.TITLE module-name comment-string	Assigns the first 15 characters in the string as an object module name and causes the string to appear on each page of the assembly listing
.TRANSFER symbol	Directs the linker to redefine the value of the global symbol for use in a shareable image
.WARN [expression] ;comment	Displays specified warning message
.WEAK symbol-list	Indicates that each of the listed symbols has the weak attribute
.WORD expression-list	Generates successive words of data; each word contains the value of the corresponding specified expression

B.2 SPECIAL CHARACTERS

The following table summarizes the VAX-11 MACRO special characters.

Table B-2
Special Characters Used in VAX-11 MACRO Statements

Character	Character Name	Function(s)
_	Underline	Character in symbol names
$	Dollar sign	Character in symbol names
.	Period	Character in symbol names, current location counter, and decimal point
:	Colon	Label terminator
=	Equal sign	Direct assignment operator and macro keyword argument terminator
	Tab	Field terminator
	Space	Field terminator
#	Number sign	Immediate addressing mode indicator

(continued on next page)

Table B-2 (Cont.)
Special Characters Used in VAX-11 MACRO Statements

Character	Character Name	Function(s)
@	At sign	Deferred addressing mode indicator and arithmetic shift operator
,	Comma	Field, operand, and item separator
;	Semicolon	Comment field indicator
+	Plus sign	Autoincrement addressing mode indicator, unary plus operator, and arithmetic addition operator
-	Minus sign	Autodecrement addressing mode indicator, unary minus operator, arithmetic subtraction operator, and line continuation indicator
*	Asterisk	Arithmetic multiplication operator
/	Slash	Arithmetic division operator
&	Ampersand	Logical AND operator
!	Exclamation point	Logical inclusive OR operator
\	Backslash	Logical exclusive OR and numeric conversion indicator in macro arguments
^	Circumflex	Unary operator indicator and macro argument delimiter
[]	Square brackets	Index addressing mode and repeat count indicators
()	Parentheses	Register deferred addressing mode indicators
<>	Angle brackets	Argument or expression grouping delimiters
?	Question mark	Created label indicator in macro arguments
'	Apostrophe	Macro argument concatenation indicator
%	Percent sign	Macro string operators

B.3 OPERATORS

B.3.1 Unary Operators

The following table summarizes the VAX-11 MACRO unary operators.

Table B-3
Unary Operators

Unary Operator	Operator Name	Example	Effect
+	Plus sign	+A	Results in the positive value of A (default)
-	Minus sign	-A	Results in the negative (2's complement) value of A
^B	Binary	^B11000111	Specifies that 11000111 is a binary number
^D	Decimal	^D127	Specifies that 127 is a decimal number
^O	Octal	^O34	Specifies that 34 is an octal number
^X	Hexadecimal	^XFCF9	Specifies that FCF9 is a hexadecimal number
^A	ASCII	^A/ABC/	Produces an ASCII string; the characters between the matching delimiters are converted to ASCII representation
^M	Register mask	^M<R3,R4,R5>	Specifies the registers R3, R4, and R5 in the register mask
^F	Floating point	^F3.0	Specifies that 3.0 is a floating-point number
^C	Complement	^C24	Produces the 1's complement value of 24 (decimal)

B.3.2 Binary Operators

The following table summarizes the VAX-11 MACRO binary operators.

Table B-4
Binary Operators

Binary Operator	Operator Name	Example	Operation
+	Plus sign	A+B	Addition
-	Minus sign	A-B	Subtraction
*	Asterisk	A*B	Multiplication
/	Slash	A/B	Division
@	At sign	A@B	Arithmetic Shift
&	Ampersand	A&B	Logical AND
!	Exclamation point	A!B	Logical inclusive OR
\	Backslash	A\B	Logical exclusive OR

B.3.3 Macro String Operators

The following table summarizes the macro string operators. These operators can be used only in macros.

Table B-5
Macro String Operators

Format	Function
%LENGTH(string)	Returns the length of the string
%LOCATE(string1,string2[,symbol])	Locates the substring string1 within string2 starting the search at the character position specified by symbol
%EXTRACT(symbol1,symbol2,string)	Extracts a substring from string that begins at character position specified by symbol1 and has a length specified by symbol2

B.4 ADDRESSING MODES

The following table summarizes the VAX-11 MACRO addressing modes.

Table B-6
Addressing Modes

Type	Addressing Mode	Format[1]	Hexadecimal Value	Description	Indexable?
General Register	Register	Rn	5	Register contains the operand	No
	Register Deferred	(Rn)	6	Register contains the address of the operand	Yes
	Autoincrement	(Rn)+	8	Register contains the address of the operand; the processor increments the register contents by the size of the operand data type	Yes
	Autoincrement Deferred	@(Rn)+	9	Register contains the address of the operand address; the processor increments the register contents by 4	Yes
	Autodecrement	-(Rn)	7	The processor decrements the register contents by the size of the operand data type; the register then contains the address of the operand	Yes
	Displacement	dis(Rn) B^dis(Rn) W^dis(Rn) L^dis(Rn)	 A C E	The sum of the contents of the register and the displacement is the address of the operand; B^, W^,and L^ indicate byte, word, and longword displacement, respectively	Yes
	Displacement Deferred	@dis(Rn) @B^dis(Rn) @W^dis(Rn) @L^dis(Rn)	 B D F	The sum of the contents of the register and the displacement is the address of the operand address; B^, W^, and L^ indicate byte, word, and longword displacement, respectively	Yes
General Register (Cont.)	Literal	#literal S^#literal	 0-3	The literal specified is the the operand; the literal is stored as a short literal	No
Program Counter	Relative	address B^address W^address L^address	 A C E	The address specified is the address of the operand; the address specified is stored as a displacement from PC; B^, W^, and L^ indicate byte, word, and longword displacement, respectively	Yes
	Relative Deferred	@address @B^address @W^address @L^address	 B D F	The address specified is the address of the operand address; the address specified is stored as a displacement from PC; B^, W^, and L^ indicate byte, word, and longword displacement, respectively	Yes
	Absolute	@#address	9	The address specified is the address of the operand; the address specified is stored as an absolute virtual address (not as a displacement)	Yes
	Immediate	#literal I^#literal	 8	The literal specified is the operand; the literal is stored as a byte, word, longword, or quadword	No

(continued on next page)

Table B-6 (Cont.)
Addressing Modes

Type	Addressing Mode	Format[1]	Hexa-decimal Value	Description	Indexable?
	General	G^address	-	The address specified is the address of the operand; if the address is defined as relocatable, the linker stores the address as a displacement from PC; if the address is defined as an absolute virtual address, the linker stores the address as an absolute value	Yes
Index	Index	base-mode[Rx]	4	The base-mode specifies the base address and the register specifies the index; the sum of the base address and the product of the contents of Rx and the size of the operand data type is the address of the operand; base-mode can be any addressing mode except register, immediate, literal, index, or branch	No
Branch	Branch	address	-	The address specified is the operand; this address is stored as a displacement to PC; branch mode can only be used with the branch instructions	No

1. Key:

Rn

Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rn.

Rx

Any general register R0 through R12. Note that the AP, FP, or SP register can be used in place of Rx. Rx cannot be the same as the Rn specified in the base-mode for certain base modes (see Section 4.3).

dis

An expression specifying a displacement.

address

An expression specifying an address.

literal

An expression, an integer constant, or a floating-point constant.

APPENDIX C

PERMANENT SYMBOL TABLE

The permanent symbol table (PST) contains the symbols that VAX-11 MACRO automatically recognizes. These symbols consist of both opcodes and assembler directives. Sections C.1 and C.2 below present the opcodes (instruction set) in alphabetical and numerical order, respectively. Appendix B (in Section B.1) presents the assembler directives.

The VAX-11 Architecture Handbook provides a detailed description of the instruction set.

C.1 OPCODES (ALPHABETIC ORDER)

Hexadecimal Value	Mnemonic	Functional Name
9D	ACBB	Add compare and branch byte
6F	ACBD	Add compare and branch D_floating
4F	ACBF	Add compare and branch F_floating
4FFD	ACBG	Add compare and branch G_floating
6FFD	ACBH	Add compare and branch H_floating
F1	ACBL	Add compare and branch long
3D	ACBW	Add compare and branch word
58	ADAWI	Add aligned word interlocked
80	ADDB2	Add byte 2 operand
81	ADDB3	Add byte 3 operand
60	ADDD2	Add D-floating 2 operand
61	ADDD3	Add D-floating 3 operand
40	ADDF2	Add F_floating 2 operand
41	ADDF3	Add F_floating 3 operand
40FD	ADDG2	Add G_floating 2 operand
41FD	ADDG3	Add G_floating 3 operand
60FD	ADDH2	Add H_floating 2 operand
61FD	ADDH3	Add H_floating 3 operand
C0	ADDL2	Add long 2 operand
C1	ADDL3	Add long 3 operand
20	ADDP4	Add packed 4 operand
21	ADDP6	Add packed 6 operand
A0	ADDW2	Add word 2 operand
A1	ADDW3	Add word 3 operand
D8	ADWC	Add with carry
F3	AOBLEQ	Add one and branch on less or equal
F2	AOBLSS	Add one and branch on less
78	ASHL	Arithmetic shift long
F8	ASHP	Arithmetic shift and round packed
79	ASHQ	Arithmetic shift quad

PERMANENT SYMBOL TABLE

Hexadecimal Value	Mnemonic	Functional Name
E1	BBC	Branch on bit clear
E5	BBCC	Branch on bit clear and clear
E7	BBCCI	Branch on bit clear and clear interlocked
E3	BBCS	Branch on bit clear and set
E0	BBS	Branch on bit set
E4	BBSC	Branch on bit set and clear
E2	BBSS	Branch on bit set and set
E6	BBSSI	Branch on bit set and set interlocked
1E	BCC	Branch on carry clear
1F	BCS	Branch on carry set
13	BEQL	Branch on equal
13	BEQLU	Branch on equal unsigned
18	BGEQ	Branch on greater or equal
1E	BGEQU	Branch on greater or equal unsigned
14	BGTR	Branch on greater
1A	BGTRU	Branch on greater unsigned
8A	BICB2	Bit clear byte 2 operand
8B	BICB3	Bit clear byte 3 operand
CA	BICL2	Bit clear long 2 operand
CB	BICL3	Bit clear long 3 operand
B9	BICPSW	Bit clear program status word
AA	BICW2	Bit clear word 2 operand
AB	BICW3	Bit clear word 3 operand
88	BISB2	Bit set byte 2 operand
89	BISB3	Bit set byte 3 operand
C8	BISL2	Bit set long 2 operand
C9	BISL3	Bit set long 3 operand
B8	BISPSW	Bit set program status word
A8	BISW2	Bit set word 2 operand
A9	BISW3	Bit set word 3 operand
93	BITB	Bit test byte
D3	BITL	Bit test long
B3	BITW	Bit test word
E9	BLBC	Branch on low bit clear
E8	BLBS	Branch on low bit set
15	BLEQ	Branch on less or equal
1B	BLEQU	Branch on less or equal unsigned
19	BLSS	Branch on less
1F	BLSSU	Branch on less unsigned
12	BNEQ	Branch on not equal
12	BNEQU	Branch on not equal unsigned
03	BPT	Break point trap
11	BRB	Branch with byte displacement
31	BRW	Branch with word displacement
10	BSBB	Branch to subroutine with byte displacement
30	BSBW	Branch to subroutine with word displacement
1C	BVC	Branch on overflow clear
1D	BVS	Branch on overflow set
FA	CALLG	Call with general argument list
FB	CALLS	Call with stack
8F	CASEB	Case byte
CF	CASEL	Case long
AF	CASEW	Case word

PERMANENT SYMBOL TABLE

Hexadecimal Value	Mnemonic	Functional Name
BD	CHME	Change mode to executive
BC	CHMK	Change mode to kernel
BE	CHMS	Change mode to supervisor
BF	CHMU	Change mode to user
94	CLRB	Clear byte
7C	CLRD	Clear D_floating
DF	CLRF	Clear F_floating
7C	CLRG	Clear G_floating
7CFD	CLRH	Clear H_floating
D4	CLRL	Clear long
7CFD	CLRO	Clear octa
7C	CLRQ	Clear quad
B4	CLRW	Clear word
91	CMPB	Compare byte
29	CMPC3	Compare character 3 operand
2D	CMPC5	Compare character 5 operand
71	CMPD	Compare D_floating
51	CMPF	Compare F_floating
51FD	CMPG	Compare G_floating
71FD	CMPH	Compare H_floating
D1	CMPL	Compare long
35	CMPP3	Compare packed 3 operand
37	CMPP4	Compare packed 4 operand
EC	CMPV	Compare field
B1	CMPW	Compare word
ED	CMPZV	Compare zero-extended field
0B	CRC	Calculate cyclic redundancy check
6C	CVTBD	Convert byte to D_floating
4C	CVTBF	Convert byte to F_floating
4CFD	CVTBG	Convert byte to G_floating
6CFD	CVTBH	Convert byte to H_floating
98	CVTBL	Convert byte to long
99	CVTBW	Convert byte to word
68	CVTDB	Convert D_floating to byte
76	CVTDF	Convert D_floating to F_floating
32FD	CVTDH	Convert D_floating to H_floating
6A	CVTDL	Convert D_floating to long
69	CVTDW	Convert D_floating to word
48	CVTFB	Convert F_floating to byte
56	CVTFD	Convert F_floating to D_floating
99FD	CVTFG	Convert F_floating to G_floating
98FD	CVTFH	Convert F_floating to H_floating
4A	CVTFL	Convert F_floating to long
49	CVTFW	Convert F_floating to word
48FD	CVTGB	Convert G_floating to byte
33FD	CVTGF	Convert G_floating to F_floating
56FD	CVTGH	Convert G_floating to H_floating
4AFD	CVTGL	Convert G_floating to long
49FD	CVTGW	Convert G_floating to word
68FD	CVTHB	Convert H_floating to byte
F7FD	CVTHD	Convert H_floating to D_floating
F6FD	CVTHF	Convert H_floating to F_floating
76FD	CVTHG	Convert H_floating to G_floating
6AFD	CVTHL	Convert H_floating to long
69FD	CVTHW	Convert H_floating to word
F6	CVTLB	Convert long to byte

PERMANENT SYMBOL TABLE

Hexadecimal Value	Mnemonic	Functional Name
6E	CVTLD	Convert long to D_floating
4E	CVTLF	Convert long to F_floating
4EFD	CVTLG	Convert long to G_floating
6EFD	CVTLH	Convert long to H_floating
F9	CVTLP	Convert long to packed
F7	CVTLW	Convert long to word
36	CVTPL	Convert packed to long
08	CVTPS	Convert packed to leading separate
24	CVTPT	Convert packed to trailing
6B	CVTRDL	Convert rounded D_floating to long
4B	CVTRFL	Convert rounded F_floating to long
4BFG	CVTRGL	Convert rounded G_floating to long
6BFD	CVTRHL	Convert rounded H_floating to long
09	CVTSP	Convert leading separate to packed
26	CVTTP	Convert trailing to packed
33	CVTWB	Convert word to byte
6D	CVTWD	Convert word to D_floating
4D	CVTWF	Convert word to F_floating
4DFD	CVTWG	Convert word to G_floating
6DFD	CVTWH	Convert word to H_floating
32	CVTWL	Convert word to long
97	DECB	Decrement byte
D7	DECL	Decrement long
B7	DECW	Decrement word
86	DIVB2	Divide byte 2 operand
87	DIVB3	Divide byte 3 operand
66	DIVD2	Divide D_floating 2 operand
67	DIVD3	Divide D_floating 3 operand
46	DIVF2	Divide F_floating 2 operand
47	DIVF3	Divide F_floating 3 operand
46FD	DIVG2	Divide G_floating 2 operand
47FD	DIVG3	Divide G_floating 3 operand
66FD	DIVH2	Divide H_floating 2 operand
67FD	DIVH3	Divide H_floating 3 operand
C6	DIVL2	Divide long 2 operand
C7	DIVL3	Divide long 3 operand
27	DIVP	Divide packed
A6	DIVW2	Divide word 2 operand
A7	DIVW3	Divide word 3 operand
38	EDITPC	Edit packed to character
7B	EDIV	Extended divide
74	EMODD	Extended modulus D_floating
54	EMODF	Extended modulus F_floating
54FD	EMODG	Extended modulus G_floating
74FD	EMODH	Extended modulus H_floating
7A	EMUL	Extended multiply
EE	EXTV	Extract field
EF	EXTZV	Extract zero-extended field
EB	FFC	Find first clear bit
EA	FFS	Find first set bit
00	HALT	Halt
96	INCB	Increment byte
D6	INCL	Increment long
B6	INCW	Increment word
0A	INDEX	Index calculation

Hexadecimal Value	Mnemonic	Functional Name
5C	INSQHI	Insert into queue at head, interlocked
5D	INSQTI	Insert into queue at tail, interlocked
0E	INSQUE	Insert into queue
F0	INSV	Insert field
17	JMP	Jump
16	JSB	Jump to subroutine
06	LDPCTX	Load program context
3A	LOCC	Locate character
39	MATCHC	Match characters
92	MCOMB	Move complemented byte
D2	MCOML	Move complemented long
B2	MCOMW	Move complemented word
DB	MFPR	Move from processor register
8E	MNEGB	Move negated byte
72	MNEGD	Move negated D_floating
52	MNEGF	Move negated F_floating
52FD	MNEGG	Move negated G_floating
72FD	MNEGH	Move negated H_floating
CE	MNEGL	Move negated long
AE	MNEGW	Move negated word
9E	MOVAB	Move address of byte
7E	MOVAD	Move address of D_floating
DE	MOVAF	Move address of F_floating
7E	MOVAG	Move address of G_floating
7EFD	MOVAH	Move address of H_floating
DE	MOVAL	Move address of long
7EFD	MOVAO	Move address of octa
7E	MOVAQ	Move address of quad
3E	MOVAW	Move address of word
90	MOVB	Move byte
28	MOVC3	Move character 3 operand
2C	MOVC5	Move character 5 operand
70	MOVD	Move D_floating
50	MOVF	Move F_floating
50FD	MOVG	Move G_floating
70FD	MOVH	Move H_floating
D0	MOVL	Move long
7DFD	MOVO	Move data
34	MOVP	Move packed
DC	MOVPSL	Move program status longword
7D	MOVQ	Move quad
2E	MOVTC	Move translated characters
2F	MOVTUC	Move translated until character
B0	MOVW	Move word
0A	MOVZBL	Move zero-extended byte to long
9B	MOVZBW	Move zero-extended byte to word
3C	MOVZWL	Move zero-extended word to long
DA	MTPR	Move to processor register
84	MULB2	Multiply byte 2 operand
85	MULB3	Multiply byte 3 operand
64	MULD2	Multiply D_floating 2 operand
65	MULD3	Multiply D_floating 3 operand
44	MULF2	Multiply F_floating 2 operand
45	MULF3	Multiply F_floating 3 operand
44FD	MULG2	Multiply G_floating 2 operand
45FD	MULG3	Multiply G_floating 3 operand

PERMANENT SYMBOL TABLE

Hexadecimal Value	Mnemonic	Functional Name
64FD	MULH2	Multiply H_floating 2 operand
65FD	MULH3	Multiply H_floating 3 operand
C4	MULL2	Multiply long 2 operand
C5	MULL3	Multiply long 3 operand
25	MULP	Multiply packed
A4	MULW2	Multiply word 2 operand
A5	MULW3	Multiply word 3 operand
01	NOP	No operation
75	POLYD	Evaluate polynomial D_floating
55	POLYF	Evaluate polynomial F_floating
55FD	POLYG	Evaluate polynomial G_floating
75FD	POLYH	Evaluate polynomial H_floating
BA	POPR	Pop registers
0C	PROBER	Probe read access
0D	PROBEW	Probe write access
9F	PUSHAB	Push address of byte
7F	PUSHAD	Push address of D_floating
DF	PUSHAF	Push address of F_floating
7F	PUSHAG	Push address of G_floating
7FFD	PUSHAH	Push address of H_floating
DF	PUSHAL	Push address of long
7FFD	PUSHAO	Push address of octa
7F	PUSHAQ	Push address of quad
3F	PUSHAW	Push address of word
DD	PUSHL	Push long
BB	PUSHR	Push registers
02	REI	Return from exception or interrupt
5E	REMQHI	Remove from queue at head, interlocked
5F	REMQTI	Remove from queue at tail, interlocked
0F	REMQUE	Remove from queue
04	RET	Return from called procedure
9C	ROTL	Rotate long
05	RSB	Return from subroutine
D9	SBWC	Subtract with carry
2A	SCANC	Scan for character
3B	SKPC	Skip character
F4	SOBGEQ	Subtract one and branch on greater or equal
F5	SOBGTR	Subtract one and branch on greater
2B	SPANC	Span characters
82	SUBB2	Subtract byte 2 operand
83	SUBB3	Subtract byte 3 operand
62	SUBD2	Subtract D_floating 2 operand
63	SUBD3	Subtract D_floating 3 operand
42	SUBF2	Subtract F_floating 2 operand
43	SUBF3	Subtract F_floating 3 operand
42FD	SUBG2	Subtract G_floating 2 operand
43FD	SUBG3	Subtract G_floating 3 operand
62FD	SUBH2	Subtract H_floating 2 operand
63FD	SUBH3	Subtract H_floating 3 operand
C2	SUBL2	Subtract long 2 operand
C3	SUBL3	Subtract long 3 operand
22	SUBP4	Subtract packed 4 operand
23	SUBP6	Subtract packed 6 operand
A2	SUBW2	Subtract word 2 operand
A3	SUBW3	Subtract word 3 operand

PERMANENT SYMBOL TABLE

Hexadecimal Value	Mnemonic	Functional Name
07	SVPCTX	Save process context
95	TSTB	Test byte
73	TSTD	Test D_floating
53	TSTF	Test F_floating
53FD	TSTG	Test G_floating
73FD	TSTH	Test H_floating
D5	TSTL	Test long
B5	TSTW	Test word
FC	XFC	Extended function call
8C	XORB2	Exclusive-OR byte 2 operand
8D	XORB3	Exclusive-OR byte 3 operand
CC	XORL2	Exclusive-OR long 2 operand
CD	XORL3	Exclusive-OR long 3 operand
AC	XORW2	Exclusive-OR word 2 operand
AD	XORW3	Exclusive-OR word 3 operand

C.2 OPCODES (NUMERIC ORDER)

HEX Value	Instruction	HEX Value	Instruction	HEX Value	Instruction	HEX Value	Instruction	HEX Value	Instruction	HEX Value	Instruction	HEX Value	Instruction
00	HALT	30	BSBW	60	ADDD2	90	MOVB	C0	ADDL2	F0	INSV	67FD	DIVH3
01	NOP	31	BRW	61	ADDD3	91	CMPB	C1	ADDL3	F1	ACBL	68FD	CVTHB
02	REI	32	CVTWL	62	SUBD2	92	MCOMB	C2	SUBL2	F2	AOBLSS	69FD	CVTHW
03	BPT	33	CVTWB	63	SUBD3	93	BITB	C3	SUBL3	F3	AOBLEQ	6AFD	CVTHL
04	RET	34	MOVP	64	MULD2	94	CLRB	C4	MULL2	F4	SOBGEQ	6BFD	CVTRHL
05	RSB	35	CMPP3	65	MULD3	95	TSTB	C5	MULL3	F5	SOBGTR	6CFD	CVTBH
06	LDPCTX	36	CVTPL	66	DIVD2	96	INCB	C6	DIVL2	F6	CVTLB	6DFD	CVTWH
07	SVPCTX	37	CMPP4	67	DIVD3	97	DECB	C7	DIVL3	F7	CVTLW	6EFD	CVTLH
08	CVTPS	38	EDITPC	68	CVTDB	98	CVTBL	C8	BISL2	F8	ASHP	6FFD	ACBH
09	CVTSP	39	MATCHC	69	CVTDW	99	CVTBW	C9	BISL3	F9	CVTLP	70FD	MOVH
0A	INDEX	3A	LOCC	6A	CVTDL	9A	MOVZBL	CA	BICL2	FA	CALLG	71FD	CMPH
0B	CRC	3B	SKPC	6B	CVTRDL	9B	MOVZBW	CB	BICL3	FB	CALLS	72FD	MNEGH
0C	PROBER	3C	MOVZWL	6C	CVTBD	9C	ROTL	CC	XORL2	FC	XFC	73FD	TSTH
0D	PROBEW	3D	ACBW	6D	CVTWD	9D	ACBB	CD	XORL3	FD	reserved	74FD	EMODH
0E	INSQUE	3E	MOVAW	6E	CVTLD	9E	MOVAB	CE	MNEGL	FE	reserved	75FD	POLYH
0F	REMQUE	3F	PUSHAW	6F	ACBD	9F	PUSHAB	CF	CASEL	FF	reserved	76FD	CVTHG
10	BSBB	40	ADDF2	70	MOVD	A0	ADDW2	D0	MOVL	32FD	CVTDH	7CFD	CLRH, CLRO
11	BRB	41	ADDF3	71	CMPD	A1	ADDW3	D1	CMPL	33FD	CVTGF	7DFD	MOVO
12	BNEQ, BNEQU	42	SUBF2	72	MNEGD	A2	SUBW2	D2	MCOML	40FD	ADDG2	7EFD	MOVAH, MOVAO
13	BEQL, BEQLU	43	SUBF3	73	TSTD	A3	SUBW3	D3	BITL	41FD	ADDG3	7FFD	PUSHAH, PUSHAO
14	BGTR	44	MULF2	74	EMODD	A4	MULW2	D4	CLRF, CLRL	42FD	SUBG2	98FD	CVTFH
15	BLEQ	45	MULF3	75	POLYD	A5	MULW3	D5	TSTL	43FD	SUBG3	99FD	CVTFG
16	JSB	46	DIVF2	76	CVTDF	A6	DIVW2	D6	INCL	44FD	MULG2	F6FD	CVTHF
17	JMP	47	DIVF3	77	reserved	A7	DIVW3	D7	DECL	45FD	MULG3	F7FD	CVTHD
18	BGEQ	48	CVTFB	78	ASHL	A8	BISW2	D8	ADWC	46FD	DIVG2		
19	BLSS	49	CVTFW	79	ASHQ	A9	BISW3	D9	SBWC	47FD	DIVG3		
1A	BGTRU	4A	CVTFL	7A	EMUL	AA	BICW2	DA	MTPR	48FD	CVTGB		
1B	BLEQU	4B	CVTRFL	7B	EDIV	AB	BICW3	DB	MFPR	49FD	CVTGW		
1C	BVC	4C	CVTBF	7C	CLRD, CLRQ, CLRG	AC	XORW2	DC	MOVPSL	4AFD	CVTGL		
1D	BVS	4D	CVTWF	7D	MOVQ	AD	XORW3	DD	PUSHL	4BFD	CVTRGL		
1E	BCC, BGEQU	4E	CVTLF	7E	MOVAD, MOVAQ, MOVAG	AE	MNEGW	DE	MOVAF, MOVAL	4CFD	CVTBG		
1F	BCS, BLSSU	4F	ACBF	7F	PUSHAD, PUSHAQ, PUSHAG	AF	CASEW	DF	PUSHAF, PUSHAL	4DFD	CVTWG		
20	ADDP4	50	MOVF	80	ADDB2	B0	MOVW	E0	BBS	4EFD	CVTLG		
21	ADDP6	51	CMPF	81	ADDB3	B1	CMPW	E1	BBC	4FFD	ACBG		
22	SUBP4	52	MNEGF	82	SUBB2	B2	MCOMW	E2	BBSS	50FD	MOVG		
23	SUBP6	53	TSTF	83	SUBB3	B3	BITW	E3	BBCS	51FD	CMPG		
24	CVTPT	54	EMODF	84	MULB2	B4	CLRW	E4	BBSC	52FD	MNEGG		
25	MULP	55	POLYF	85	MULB3	B5	TSTW	E5	BBCC	53FD	TSTG		
26	CVTTP	56	CVTFD	86	DIVB2	B6	INCW	E6	BBSSI	54FD	EMODG		
27	DIVP	57	reserved	87	DIVB3	B7	DECW	E7	BBCCI	55FD	POLYG		
28	MOVC3	58	ADAWI	88	BISB2	B8	BISPSW	E8	BLBS	56FD	CVTGH		
29	CMPC3	59	reserved	89	BISB3	B9	BICPSW	E9	BLBC	60FD	ADDH2		
2A	SCANC	5A	reserved	8A	BICB2	BA	POPR	EA	FFS	61FD	ADDH3		
2B	SPANC	5B	reserved	8B	BICB3	BB	PUSHR	EB	FFC	62FD	SUBH2		
2C	MOVC5	5C	INSQHI	8C	XORB2	BC	CHMK	EC	CMPV	63FD	SUBH3		
2D	CMPC5	5D	INSQTI	8D	XORB3	BD	CHME	ED	CMPZV	64FD	MULH2		
2E	MOVTC	5E	REMQHI	8E	MNEGB	BE	CHMS	EE	EXTV	65HD	MULH3		
2F	MOVTUC	5F	REMQTI	8F	CASEB	BF	CHMU	EF	EXTZV	66FD	DIVH2		

APPENDIX D

HEXADECIMAL/DECIMAL CONVERSION

Table D-1 lists the decimal value for each possible hexadecimal value in each byte of a longword. The following sections contain instructions to use the table to convert hexadecimal numbers to decimal and vice versa.

D.1 HEXADECIMAL TO DECIMAL

For each integer position of the hexadecimal value, locate the corresponding column integer and record its decimal equivalent in the conversion table. Add the decimal equivalent to obtain the decimal value.

For example:

```
D0500AD0 (16)    =       ?(10)

D0000000         =  3,489,660,928
  500000         =      5,242,880
     A00         =          2,560
      D0         =            208
________            _____________

D0500AD0         =  3,494,904,576
```

D.2 DECIMAL TO HEXADECIMAL

Step 1: locate in the conversion table the largest decimal value that does not exceed the decimal number to be converted. Step 2: record the hexadecimal equivalent followed by the number of 0s that corresponds to the integer column minus 1. Step 3: subtract the table decimal value from the decimal number to be converted. Step 4: repeat steps 1 through 3 until the subtraction balance equals 0. Add the hexadecimal equivalents to obtain the hexadecimal value.

Example:

```
        22,466 (10)      =  ?(16)

        20,480           =  5000        22,466
         1,792           =   700       -20,480
           192           =    C0       -------
             2           =     2         1,986
        ------              ----       - 1,792
        22,466           =  57C2       -------
                                           194
                                       -   192
                                       -------
                                             2
                                       -     2
                                       -------
                                             0
```

D.3 POWERS OF 2 AND 16

This section lists the decimal values of powers of 2 and 16. These values are often useful in converting decimal numbers to hexadecimal.

Powers of 2

2**n	n
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20
2097152	21
4194304	22
8388608	23
16777216	24

Powers of 16

16**n	n
1	0
16	1
256	2
4096	3
65536	4
1048576	5
16777216	6
268435456	7
4294967296	8
68719476736	9
1099511627776	10
17592186044416	11
281474976710656	12
4503599627370496	13
72057594037927936	14
1152921504606846976	15

Table D-1
Hexadecimal/Decimal Conversion

HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,879,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,643	8	134,217,728	8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,919,104	9	150,994,944	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,658,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
BYTE				BYTE				BYTE				BYTE			
WORD								WORD							
LONGWORD															

A

^A operator, 3-11, 3-13
Absolute,
  index mode, 4-15, 4-17
  mode, 4-3, 4-13, 4-14
  program sections, 5-46, 5-48
ABS program section attribute, 5-48
Accuracy of floating-point numbers, 3-4
Addition, 3-15
Address data,
  initializing memory with, 5-3
  reserving memory for, 5-9
.ADDRESS directive, 5-3
Address, starting, 5-21, 5-22
Address, transfer, 5-20, 5-21
Addressing modes, 2-3, 4-1 through 4-18
  summary of, 4-2 through 4-4, B-11 through B-13
.ALIGN directive, 5-4, 5-5
Alignment,
  data, 5-4, 5-5
  location counter, 5-4, 5-5, 5-24, 5-41
  program section, 5-47, 5-50
AMA argument, 5-18
AND operator, 3-16
AP register, 3-5
Argument,
  concatenation in macros, 6-6
  macro, 6-1 through 6-8
  pointer, 3-5
Arithmetic shift,
  operator, 3-16
Arithmetic trap enable, 3-13, 3-14
.ASCIC directive, 5-7
.ASCID directive, 5-8
ASCII character set, A-1
.ASCII directive, 5-7
ASCII,
  hexadecimal conversion, A-1
  operator, 3-11, 3-13
  strings, 3-11, 3-13, 5-6 through 5-9
  string storage, 5-6 through 5-9
.ASCIx storage directives, 5-6 through 5-9
.ASCIZ directive, 5-8
Assembler directives, 2-3, 5-1 through 5-67, 6-1 through 6-29
  summary of, 5-1, 5-2, 6-2, B-1 through B-7
Assembler functions, 5-17 through 5-19
Assigning a value, 3-17
Assignment statements, 3-17, 3-18
Attributes, program section, 5-46 through 5-50
Autodecrement index mode, 4-15, 4-17
Autodecrement mode, 4-7
Autoincrement deferred index mode, 4-15, 4-16, 4-17
Autoincrement deferred mode, 4-6, 4-7
Autoincrement index mode, 4-15, 4-16, 4-17
Autoincrement mode, 4-6

B

B^ displacement specifier, 4-8, 4-9, 4-12
^B unary operator, 3-11, 3-12
Base mode, 4-15, 4-17
Binary operators, 3-15 through 3-17, B-10
Binary radix, 3-11, 3-12
Blank lines, 2-2
.BLKA directive, 5-9, 5-10
.BLKB directive, 5-9, 5-10
.BLKD directive, 5-9, 5-10
.BLKF directive, 5-9, 5-10
.BLKG directive, 5-9, 5-10
.BLKH directive, 5-9, 5-10
.BLKL directive, 5-9, 5-10
.BLKO directive, 5-9, 5-10
.BLKQ directive, 5-9, 5-10
.BLKW directive, 5-9, 5-10
.BLKx directives, 5-9, 5-10
Block labels, 3-7, 3-8
Block storage directives, 5-9, 5-10
Branch addresses, 3-8
Branch instructions, 4-18
Branch mode, 4-18
Byte data,
  initializing memory with, 5-11, 5-57
  reserving memory for, 5-9, 5-10,
.BYTE directive, 5-11

C

^C operator, 3-15
Call instruction, 5-21, 5-22
Character,
  indefinite repeat block, 6-16, 6-17

Character, (Cont.)
  separating, 3-3
  set, 3-1, 3-2
  set, ASCII, A-1
  special, 3-1, 3-2, B-7, B-8
  strings, 3-12, 3-13, 5-6 through 5-8
Characters, counting, 6-9, 6-25
Combining arguments in macros, 6-6
Comment field, 2-1, 2-4
Complement operator, 3-15
Concatenated program sections, 5-48
Concatenating arguments in macros, 6-6
Conditional assembly blocks, 5-30 through 5-36
  controlling listing of, 5-55, 5-56
  one line block, 5-36
  subconditionals, 5-33 through 5-35
Condition tests, 5-30, 5-31
CON program section attribute, 5-48
Continuation lines, 2-2
Continuing program sections, 5-46
Controlling listings, 5-55, 5-56
Counted ASCII string storage, 5-7
Counter, current location, 3-18
Counting characters, 6-9, 6-25
Counting macro arguments, 6-24
Counts, repeat, 3-10, 5-11, 5-38, 5-57, 5-58, 5-67
Created local labels, 6-7, 6-8
.CROSS directive, 5-12, 5-13
Cross reference listing, 5-12, 5-13
Current location counter, 3-18

D

^D unary operator, 3-11, 3-12
Data alignment, 5-4, 5-5
Data, reserving memory for, 5-9
Data, initializing memory with,
  address, 5-3
  ASCII, 5-6 through 5-8
  byte, 5-11, 5-57
  D_floating, 5-16
  Double precision, 5-16
  F_floating, 5-25
  floating-point, 5-16, 5-25, 5-26, 5-28
  G_floating, 5-26
  H_floating, 5-28
  longword, 5-38
  octaword, 5-41
Data, initializing memory with, (Cont.)
  packed decimal, 5-44
  quadword, 5-51
  signed, 5-57, 5-58, 5-59
  word, 5-58, 5-67
DBG argument, 5-18
.DEBUG directive, 5-14
Debugging information, 5-14, 5-18
Decimal/hexadecimal conversion, D-1 through D-4
Decimal radix, 3-11, 3-12
Decimal strings, 3-4, 3-5, 5-44
.DEFAULT directive, 5-15
Default program sections, 5-46, 5-50
Default radix, 3-12
Default values of macro arguments, 6-3
Deferred mode,
  autoincrement, 4-6, 4-7
  displacement, 4-9
  register, 4-5
  relative, 4-12
Defining,
  labels, 2-2
  macros, 6-19, 6-20
  opcodes, 5-42, 5-43
Degree of precision, 3-4
Deleting a macro, 6-22
Delimited ASCII strings, 5-6
Delimiters in macro arguments, 6-4 through 6-6
Descriptors, string, 5-8
D_floating data,
  initializing memory with, 5-16
  reserving memory for, 5-9, 5-10
.D_FLOATING directive, 5-16
Direct assignment statements, 3-17, 3-18
Directives, 2-3, 3-5, 5-1 through 5-67, 6-1 through 6-29
  summary of, 5-1, 5-2, 6-2, B-1 through B-7
.DISABLE directive, 5-17 through 5-19
  LOCAL_BLOCK attribute, 3-8
Disabling assembler functions, 5-17 through 5-19
Displacement,
  controlling default, 5-15
  deferred index mode, 4-15 through 4-17
  deferred mode, 4-9
  index mode, 4-15 through 4-17
  mode, 4-7, 4-8
  specifier, 4-8, 4-9, 4-12
Division, 3-15
Documenting a program, 2-4

.DOUBLE directive, 5-16
Double precision, 3-4, 5-16
Double-precision data,
  initializing memory with, 5-16
  reserving memory for, 5-9, 5-10
.DSABL directive, 5-17 through 5-19
DV arithmetic trap enable, 3-13, 3-14

E

.ENABL directive, 5-17 through 5-19
.ENABLE directive, 5-17 through 5-19
  LOCAL_BLOCK attribute, 3-8
Enabling assembler functions, 5-17 through 5-19
.END directive, 5-20
.ENDC directive, 5-20, 5-30 through 5-35
Ending,
  conditional assembly blocks, 5-20, 5-30 through 5-35
  macro definitions, 6-13, 6-19, 6-20
  modules, 5-20
  repeat range definitions, 6-13, 6-14 through 6-17
.ENDM directive, 6-13, 6-19, 6-20
.ENDR directive, 6-13 through 6-17
.ENTRY directive, 5-21
Entry mask, 3-13, 3-14, 5-20, 5-21, 5-39
.ERROR directive, 5-23
Exclusive OR operator, 3-17
Executable program sections, 5-46 through 5-50
EXE program section attribute, 5-48
Expanding a macro, 6-1, 6-2
Exponent, 3-4
Expressions, 3-9, 3-10
  evaluation of, 3-9
  floating point, 3-14
  restrictions on, 3-10
Extended precision, 3-4, 5-28
.EXTERNAL directive, 5-24
External symbols, 3-7, 5-24, 5-27, 5-66
%EXTRACT macro string operator, 6-11, 6-12
.EXTRN directive, 5-24
.EVEN directive, 5-24
Exiting a macro, 6-13, 6-19, 6-20

F

^F operator, 3-14, 3-15
Factors, repetition, 3-10, 5-11, 5-38, 5-57, 5-58, 5-67
Field,
  comment, 2-1, 2-4
  label, 2-1 through 2-3
  operand, 2-1, 2-3, 2-4
  operator, 2-1, 2-3
F_floating data,
  initializing memory with, 5-25
  reserving memory for, 5-9, 5-10
.F_FLOATING directive, 5-25
.FLOAT directive, 5-25
Floating-point data,
  initializing memory with, 5-16, 5-25, 5-26, 5-28
  reserving memory for, 5-9, 5-10
Floating-point expressions, 3-14
Floating-point numbers, 3-3, 3-4, 3-14, 3-15
  format of, 3-4
  rounding of, 5-17, 5-19
  truncation of, 5-17, 5-19
Floating-point operator, 3-14
Floating-point short literals, 4-10
Format, statement, 2-1 through 2-4
Formatting with tabs, 2-1, 2-2
FP register, 3-6, 3-14
FPT attribute, 5-19
Frame pointer, 3-6
Functions, assembler, 5-17 through 5-19

G

GBL argument, 5-18, 5-19
GBL program section attribute, 5-48
General mode, 4-14
General registers, 3-5
General register modes, 4-1 through 4-11
G_floating data,
  initializing memory with, 5-26
  reserving memory for, 5-9, 5-10
.G_FLOATING directive, 5-26
GLOBAL argument, 5-18, 5-19
Global attribute, 5-48, 5-49
.GLOBAL directive, 5-27
Global program sections, 5-48
Global symbols, 2-2, 3-7, 3-17, 5-18, 5-19, 5-24, 5-27, 5-66
  defining, 2-2
  weak, 5-66
.GLOBL directive, 5-27

H

Hexadecimal/ASCII conversion, A-1
Hexadecimal/decimal conversion, D-1 through D-4
Hexadecimal radix, 3-11, 3-12
H_floating data,
  initializing memory with, 5-28
  reserving memory for, 5-9, 5-10
.H_FLOATING directive, 5-28

I

I^ addressing mode, 4-14, 4-15
.IDENT directive, 5-29
Identifying a module, 5-29, 5-61
.IF directive, 5-30 through 5-32
.IF_FALSE directive, 5-33 through 5-35
.IF_TRUE directive, 5-33 through 5-35
.IF_TRUE_FALSE directive, 5-33 through 5-35
.IFF directive, 5-33 through 5-35
.IFT directive, 5-33 through 5-35
.IFTF directive, 5-33 through 5-35
.IFx directives, 5-33 through 5-35
.IIF directive, 5-36
Immediate conditional block, 5-36
Immediate mode, 4-13, 4-14
Inclusive OR operator, 3-15, 3-17
Indefinite repeat blocks, 6-14, 6-15
Indefinite repeat character blocks, 6-16, 6-17
Index mode, 4-15 through 4-17
Initializing memory with
  address data, 5-3
  ASCII data, 5-6 through 5-8
  byte data, 5-11, 5-57
  D_floating data, 5-16
  floating-point data, 5-16, 5-25, 5-26, 5-28
  G_floating data, 5-26
  H_floating data, 5-28
  longword data, 5-38
  octaword data, 5-41
  packed data, 5-44
  quadword data, 5-51
  word data, 5-58, 5-67
Instructions, 1-1, 2-3, 3-5, C-1 through C-8
  redefining, 5-42, 5-43, 6-20
Integer expressions, 3-9, 3-10
Integers, 3-3
Internal symbols, 2-2, 3-7, 3-17
.IRP directive, 6-14, 6-15
.IRPC directive, 6-16, 6-17
IV arithmetic trap enable, 3-14

K

Keyword arguments in macros, 6-3, 6-4

L

L^ displacement specifier, 4-8, 4-9, 4-12, 4-13
Label,
  defining a, 2-2
  field, 2-1, 2-2
  local, 3-7, 3-8
  names, 2-2
  terminator, 2-2
LCL program section attribute, 5-48
%LENGTH macro string operator, 6-9
Length of source line, 2-1
Lexical operators, 6-8 through 6-12
.LIBRARY directive, 6-18
Lines, continuation, 2-2
.LIST directive, 5-37, 5-55, 5-56
Listing,
  control of, 5-44, 5-55, 5-56
  cross reference, 5-13, 5-14
  table of contents, 5-59
Literal mode, 4-10 through 4-12
Literals, short, 4-10 through 4-12
Local label block,
  delimiters, 3-8
  disabling, 3-8, 5-17, 5-18
  enabling, 3-8, 5-17, 5-18
  restoring, 5-53
  saving, 5-54
Local labels, 3-7, 3-8, 5-17, 5-18
  created, 6-7, 6-8
Local program sections, 5-48, 5-49
%LOCATE macro string operator, 6-10, 6-11
Location counter, 3-18
  alignment, 5-4, 5-5, 5-24, 5-41
Logical AND operator, 3-16
Logical exclusive OR operator, 3-17
Logical inclusive OR operator, 3-17
.LONG directive, 5-38
Longword data,
  initializing memory with, 5-38
  reserving memory for, 5-9, 5-10
LSB argument, 3-8, 5-18

M

^M operator, 3-13, 3-14
Machine instructions, 1-1
.MACRO directive, 6-19, 6-20
Macros, 6-1 through 6-29
  arguments in, 6-1 through 6-8
  calls to, 2-3
  controlling listing of, 5-55, 5-56
  definitions of, 6-19, 6-20
  deletion of, 6-22
  exiting from, 6-23
  expanding, 6-1, 6-2
  libraries containing, 6-18
  maximum line size, 2-2
  names of, 3-6, 6-13, 6-19, 6-20
  redefining, 6-19, 6-20
  string operators in, 6-8 through 6-12
.MASK directive, 5-21, 5-39
Mask operator, 3-13, 3-14
Mask, register save, 3-13, 3-14, 5-21, 5-39
.MCALL directive, 6-21
.MDELETE directive, 6-22
Messages, printing assembly, 5-23, 5-45, 5-65
.MEXIT directive, 6-23
Mnemonic instructions, 3-5, C-1 through C-8
Modes, addressing, 2-3, 4-1 through 4-18
  summary of, 4-2 through 4-4, B-11 through B-13
Module, identifying, 5-29, 5-61
Multiplication, 3-15

N

Names,
  macro, 3-6, 6-13, 6-19, 6-20
  module, 5-61
  register, 3-5
  symbol, 3-6
.NARG directive, 6-24
.NCHR directive, 6-25
Negative numbers, 3-3
.NLIST directive, 5-39, 5-55, 5-56
.NOCROSS directive, 5-12, 5-13, 5-40
NOEXE program section attribute, 5-48
NOPIC program section attribute, 5-48
.NOSHOW directive, 5-39, 5-55, 5-56
NOSHR program section attributes, 5-48
NOWRT program section attribute, 5-49
.NTYPE directive, 6-26
Number of macro arguments, 6-1, 6-24
Numbers, 3-3
  floating point, 3-3, 3-4, 3-14, 3-15, 5-16, 5-25, 5-26, 5-28
  integer, 3-3
  packed decimal, 3-4, 3-5, 5-44
Numeric control operators, 3-14, 3-15

O

^O unary operator, 3-11, 3-12
.OCTA directive, 5-41
Octal radix, 3-11, 3-12
Octaword data,
  initializing memory with, 5-28
  reserving memory for, 5-9, 5-10
.ODD directive, 5-41
Opcodes, C-1 through C-8
  defining, 5-42, 5-43
  redefining, 5-42, 6-20
.OPDEF directive, 5-42, 5-43
Operand,
  descriptors, 5-43
  field, 2-1, 2-3
  generation directives, 5-52
  types, 6-26
Operator,
  binary, 3-15 through 3-17, B-9
  field, 2-1, 2-3
  macro string, 6-8 through 6-12, B-10
  unary, 3-10 through 3-15, B-8, B-10
OR operators, 3-17
Overlaid program sections, 5-49
OVR program section attribute, 5-49

P

Packed decimal strings, 3-4, 3-5, 5-44
.PACKED directive, 5-44
.PAGE directive, 5-44
Page ejection, 5-44
Passing numeric values in macros, 6-7
PC register, 3-6
Permanent symbols, 3-5, C-1
PIC program section attribute, 5-49
Position-independent code, 5-49
Precision of floating-point numbers, 3-4

.PRINT directive, 5-45
Printing assembly messages, 5-23, 5-45, 5-65
Privileged shareable image attribute, 5-49
Program counter, 3-6
Program counter modes, 4-11 through 4-14
Program section attributes, 5-48, 5-49
  default, 5-49
Program sections, 5-46 through 5-50, 5-53, 5-54

Q

.QUAD directive, 5-51
Quadword data,
  initializing memory with, 5-51
  reserving memory for, 5-9, 5-10

R

Radix control, 3-11, 3-12
Radix default, 3-11
Radix operators, 3-11, 3-12
  in macro arguments, 6-5
Real numbers, 3-3, 3-4
Redefining
  instructions, 5-42, 6-20
  macros, 6-19, 6-20
  opcodes, 5-42, 6-20
.REFn directive, 5-52
Register,
  deferred index mode, 4-15 through 4-17
  deferred mode, 4-5
  mask operator, 3-13, 3-14
  mode, 4-5
  names, 3-5, 3-6
  save mask, 3-13, 3-14, 5-21, 5-39
Relative,
  default displacement, 5-15
  deferred index mode, 4-15, 4-17
  deferred mode, 4-12
  index mode, 4-15, 4-17
  mode, 4-11, 4-12
Relocatable program sections, 5-49, 5-50
REL program section attribute, 5-49
Repeat blocks, 6-28, 6-29
  character, indefinite repeat, 6-16, 6-17
  controlling listing of, 5-55, 5-56
  indefinite, 6-14, 6-15
Repeat counts, 3-10, 5-11, 5-38, 5-57, 5-58, 5-67
.REPEAT directive, 6-28, 6-29
Repeating a block of code, 6-28, 6-29
Repetition factors, 3-10, 5-11, 5-38, 5-57, 5-58, 5-67
.REPT directive, 6-28, 6-29
.RESTORE directive, 5-53
.RESTORE_PSECT directive, 5-53
Restoring a program section, 5-53
Reserved bits in entry mask, 3-14, 5-21
Reserving storage, 5-9
Rounding floating-point numbers, 5-19

S

S^ addressing mode, 4-10 through 4-12
.SAVE directive, 5-54
.SAVE_PSECT directive, 5-54
Saving a program section, 5-54
Saving local label block, 5-54
.SBTTL directive, 5-60
Sections, program, 5-46 through 5-50, 5-53, 5-54
Separating characters, 3-3
Shareable images, 5-62 through 5-64
Shareable program sections, 5-49
Shift operator, arithmetic, 3-16
Short literals, 4-10, 4-11
.SHOW directive, 5-55, 5-56
SHR program section attribute, 5-49
.SIGNED_BYTE directive, 5-57
Signed data storage, 5-57 through 5-59
.SIGNED_WORD directive, 5-58, 5-59
Single precision, 3-4, 5-25
Single-precision data,
  initializing memory with, 5-25
  reserving memory for, 5-9, 5-10
Source lines,
  blank, 2-2
  continuing, 2-2
  format of, 2-1
  length of, 2-1
SP register, 3-6
Special characters, 3-1, 3-2, B-7, B-8
Stack pointer, 3-6
Starting address, 5-21, 5-22
Statement format, 2-1 through 2-4
Storage, reserving, 5-9
  ASCII, 5-6 through 5-8
  block, 5-9

Storing,
  address, 5-3
  ASCII, 5-6 through 5-8
  byte, 5-11, 5-57
  D_floating, 5-16
  double-precision, 5-16
  F_floating, 5-25
  floating-point, 5-25
  G_floating, 5-26
  H_floating, 5-28
  longword, 5-37
  octaword, 5-28
  packed decimal, 5-42
  signed, 5-57, 5-58
  quadword, 5-51
  word, 5-58, 5-67
String,
  arguments in macros, 6-4 through 6-6
  ASCII, 3-12, 3-13, 5-6
  descriptors, 5-8
  operators, 6-8 through 6-12
  packed decimal, 3-4, 3-5, 5-44
Subconditional assembly blocks, 5-33 through 5-35
.SUBTITLE directive, 5-60
Subtraction, 3-15
Suppressing symbol table listing, 5-17, 5-18
Symbols, 3-5, 3-17
  external, 3-7, 5-24, 5-27, 5-66
  global, 2-2, 3-7, 3-17, 5-18, 5-19, 5-24, 5-27, 5-66
  internal, 3-7
  names of, 3-6
  permanent, 3-5, C-1
  undefined, 5-17 through 5-19
  user-defined, 2-2, 3-6

T

Tab formatting, 2-1, 2-2
Table of contents, listing, 5-60
TBK argument, 5-18
Temporary labels, 3-7, 3-8
Terms, 3-9
Testing conditions, 5-30 through 5-32
Textual operators, 3-12 through 3-14
.TITLE directive, 5-61
Traceback information, 5-18
.TRANSFER directive, 5-62 through 5-64
Trap enable, arithmetic, 3-13, 3-14
Truncating floating-point number, 5-19
Type of operand in macros, 6-24

U

Unary operators, 3-11 through 3-15, B-9
  in macro arguments, 6-5
  summary of, 3-11, B-9
Undefined symbols, 5-17 through 5-19
User-defined program sections, 5-46 through 5-50
User-defined symbol, 2-2
User-generated
  errors, 5-23
  messages, 5-45
  opcodes, 5-42, 5-43
  operands, 5-52
  warnings, 5-65

V

Value, passing arguments by, 6-7
VEC program section attribute, 5-49
Vector, transfer, 5-62 through 5-64
Version number, 5-29

W

W^ displacement specifier, 4-8, 4-9, 4-12, 4-13
.WARN directive, 5-65
Warning directive, 5-65
.WEAK directive, 5-66
Weak symbols, 5-66
Word data,
  initializing memory with, 5-58, 5-67
  reserving memory for, 5-9, 5-10
.WORD directive, 5-67
Write protecting program sections, 5-46, 5-49, 5-50
WRT program section attribute, 5-49

X

^X unary operator, 3-12

Z

Zero terminated ASCII string, 5-8

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- [ ] Assembly language programmer
- [ ] Higher-level language programmer
- [ ] Occasional programmer (experienced)
- [ ] User with little programming experience
- [ ] Student programmer
- [ ] Other (please specify)______

Name______ Date______

Organization______

Street______

City______ State______ Zip Code or Country______

Do Not Tear - Fold Here and Tape

digital

No Postag
Necessary
if Mailed in
United Stat

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

BSSG PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

Do Not Tear - Fold Here