VAX MACRO and Instruction Set Reference Manual

Order Number: AA–LA89B–TE

June 1990

This document describes the features of the VAX MACRO instruction set and assembler. It includes a detailed description of MACRO directives and instructions, as well as information about MACRO source program syntax.

Revision/Update Information:	This manual supersedes the *VAX MACRO and Instruction Set Reference Manual*, Version 5.2
Software Version:	VMS Version 5.4

digital equipment corporation
maynard, massachusetts

June 1990

CDA	DEQNA	MicroVAX	VAX RMS
DDIF	Desktop–VMS	PrintServer 40	VAXserver
DEC	DIGITAL	Q-bus	VAXstation
DECdtm	GIGI	ReGIS	VMS
DECnet	HSC	ULTRIX	VT
DECUS	LiveLink	UNIBUS	XUI
DECwindows	LN03	VAX	
DECwriter	MASSBUS	VAXcluster	digital™

ZK4515

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by Digital. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use Digital-supported devices, such as the LN03 laser printer and PostScript printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

Contents

Preface

This manual describes the VAX MACRO language and the VAX instruction set. It includes the format and function of each feature of the language. The *VAX Architecture Reference Manual* describes the instruction set in greater detail.

Intended Audience

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

Document Structure

This manual is divided into two parts, each of which is subdivided into several chapters.

Part I describes the VAX MACRO language.

- Chapter 1 introduces the features of the VAX MACRO language.
- Chapter 2 describes the format used in VAX MACRO source statements.
- Chapter 3 describes the following components of VAX MACRO source statements:
  - Character set
  - Numbers
  - Symbols
  - Local labels
  - Terms and expressions
  - Unary and binary operators
  - Direct assignment statements
  - Current location counter
- Chapter 4 describes the arguments and string operators used with macros.
- Chapter 5 summarizes and gives examples of using the VAX MACRO addressing modes.
- Chapter 6 describes the VAX MACRO general assembler directives and the directives used in defining and expanding macros.

Part II describes the VAX data types, the instruction and addressing mode formats, and the instruction set.

- Chapter 7 summarizes the terminology and conventions used in the descriptions in Part II.
- Chapter 8 describes the basic VAX architecture, including the following:
  - Address space
  - Data types
  - Processor status longword
  - Permanent exception enables
  - Instruction and addressing mode formats
- Chapter 9 describes the native-mode instruction set. The instructions are divided into groups according to their function and are listed alphabetically within each group.
- Chapter 10 describes the extension to the VAX architecture for integrated vector processing.

This manual also contains the following five appendixes:

- Appendix A lists the ASCII character set used in VAX MACRO programs.
- Appendix B gives rules for hexadecimal/decimal conversion.
- Appendix C summarizes the general assembler and macro directives (in alphabetical order), special characters, unary operators, binary operators, macro string operators, and addressing modes.
- Appendix D lists the permanent symbols (instruction set) defined for use with VAX MACRO.
- Appendix E describes the exceptions (traps and faults) that may occur during instruction execution.

Associated Documents

The following documents are relevant to VAX MACRO programming:

- *VAX Architecture Reference Manual*
- *VMS DCL Dictionary*
- The descriptions of the VMS Linker and Symbolic Debugger in:
  - *VMS Linker Utility Manual*
  - *VMS Debugger Manual*
- *Introduction to VMS System Routines*
- *VMS Run-Time Library Routines Volume*

Conventions

The following conventions are used in this manual:

Ctrl/x	A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
$\boxed{\text{Return}}$	In examples, a key name is shown enclosed in a box to indicate that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	In examples, a horizontal ellipsis indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
[]	In format descriptions, brackets indicate that whatever is enclosed within the brackets is optional; you can select none, one, or all of the choices. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
{}	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason. Boldface text is also used to show user input in online versions of the book.
UPPERCASE TEXT	Uppercase letters indicate that you must enter a command (for example, enter OPEN/READ), or they indicate the name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege.
-	Hyphens in coding examples indicate that additional arguments to the request are provided on the line that follows.
numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

VAX MACRO Language

Part I provides an overview of the features of the VAX MACRO language. It includes an introduction to the structure and components of VAX MACRO source statements. Part I also contains a detailed discussion of the VAX MACRO addressing modes, general assembler directives, and macro directives.

1 Introduction

VAX MACRO is an assembly language for programming VAX computers using the VMS operating system. Source programs written in VAX MACRO are translated into object (or binary) code by the VAX MACRO assembler, which produces an object module and, optionally, a listing file. The features of the language are introduced in this chapter.

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

- VAX 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 instruction set is summarized in Appendix D of this volume and is described in detail in Chapter 9. **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 operations:

- Store data or reserve memory for data storage
- Control the alignment of parts of the program in memory
- Specify the methods of accessing the sections of memory in which the program will be stored
- Specify the entry point of the program or a 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 you to perform the following operations:

- Repeat identical or similar sequences of source statements throughout a program without rewriting those sequences
- Use string operators to manipulate and test the contents of source statements

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

2 VAX MACRO Source Statement Format

A source program consists of a sequence of source statements that the assembler interprets and processes, one at a time, 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, to ensure that the source line fits (with its binary expansion) on one line in the listing file, no line should exceed 80 characters.

VAX MACRO statements can consist of up to four fields, as follows:

- Label field—symbolically defines a location in a program.
- Operator field—specifies the action to be performed by the statement; can be an instruction, an assembler directive, or a macro call.
- Operand field—contains the instruction operands or the assembler directive arguments or the macro arguments.
- Comment field—contains a comment that explains the meaning of the statement; 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 begins 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 and is a Digital convention.

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,^M<>              ; Beginning of routine
        CLRL    R0                      ; Clear register
LABT:   SUBL3   #10,4(AP),R2            ; Subtract 10
LAB2:   BRB     CONT                    ; Branch to another routine
```

Continue a single statement 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$AL_VECTOR,-        ; Save boot driver
                RPB$L_IOVEC(R7)
```

VAX MACRO treats the preceding 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. Do not continue permanent and user-defined symbol names 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 is terminated at that character. Section 3.3 describes symbols in detail.

Note that when a statement occurs in a macro definition (see Chapter 4 and Chapter 6), the statement cannot contain more than 1000 characters.

Blank lines are legal, but they 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 where the label occurs. The user-defined symbol name can be up to 31 characters long and can contain any alphanumeric character and the underscore (_), dollar sign ($), and period (.) characters. See Section 3.3.2 for a description of 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.

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

If a label extends past column 7, place it on a line by itself so that the following operator field can start in column 9 of the next line.

The following example illustrates some of the ways you can define labels:

```
EXP:    .BLKL   50      ; Table stores expected values
DATA::  .BLKW   25      ; Data table accessed by store
                        ;   routine in another module
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    ; Go to exit routine
```

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 an instruction, an assembler directive, or a macro call.

When the operator is an instruction, VAX MACRO generates the binary code for that instruction in the object module. The binary codes are listed in Appendix D; the instruction set is described in Chapter 9. When the operator is a directive, VAX MACRO performs certain control actions or processing operations during source program assembly. The assembler directives are described in Chapter 6. When the operator is a macro call, VAX MACRO expands the macro. Macro calls are described in Chapter 4 and in Chapter 6 (.MACRO directive).

Use either a space or a tab character to terminate the operator field; however, the tab is the recommended termination character.

2.3 Operand Field

The operand field can contain operands for instructions or arguments for either 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 5. The operand field for a specific instruction must contain the number of operands required by that instruction. See Chapter 9 for descriptions of the instructions and their operands.

Arguments for a directive must meet the format requirements of that directive. Chapter 6 describes 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 must be separated by commas (,). VAX 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 for a discussion of expressions). 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. You can cause user-written messages to be displayed during assembly by the .ERROR, .PRINT, and .WARN directives (see descriptions in Chapter 6).

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

To continue a lengthy comment to the next line, write the comment on the next line and precede it with another semicolon. 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.

Write the text of a comment to convey the meaning rather than the action of the statement. The instruction MOVAL BUF_PTR_1,R7, for example, 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
```

3 Components of MACRO Source Statements

This chapter describes the following components of VAX MACRO source statements:

- Character set
- Numbers
- Symbols
- Local labels
- Terms and expressions
- Unary and binary operators
- Direct assignment statements
- Current location counter

3.1 Character Set

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

- The letters of the alphabet, A to Z, uppercase and lowercase. Note that the assembler considers lowercase letters equivalent to uppercase letters except when they appear in ASCII strings.
- The digits 0 to 9.
- The special characters listed in Table 3–1.

Table 3–1 Special Characters Used in VAX MACRO Statements

Character	Character Name	Function
_	Underscore	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 3–1 (Cont.) Special Characters Used in VAX MACRO Statements

Character	Character Name	Function
@	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 point
\	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 local 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 MACRO.

Table 3–2 Separating Characters in VAX MACRO Statements

Character	Character Name	Usage
(space) *(tab)*	Space or tab	Separator between statement fields. Spaces within expressions 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 is legal for the current radix.

VAX 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 –2,147,483,648 to +2,147,483,647 for signed data or in the range of 0 to 4,294,967,295 for unsigned data.

Negative numbers must be preceded by a minus sign; VAX MACRO translates such numbers into two'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 6) 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 (F_FLOATING). Section 3.6 and Section 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 to 9.

The decimal point can appear anywhere to the right of the first digit. Note that a floating-point number cannot start with a decimal point because VAX 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 6) control whether single-precision floating-point numbers are rounded or truncated. Double-precision and extended-precision floating-point numbers are always rounded.

Section 8.2.6, Section 8.2.7, Section 8.2.8, and Section 8.2.9 describe 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 6).

Format

```
snn
```

s
An optional sign.

nn
A string containing up to 31 decimal digits in the range of 0 to 9.

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

Section 8.2.14 describes the internal format of packed decimal strings.

3.3 Symbols

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

3.3.1 Permanent Symbols

Permanent symbols consist of instruction mnemonics (see Appendix D), VAX MACRO directives (see Chapter 6), and register names. You need not define instruction mnemonics and directives before you use them in the operator field of a VAX MACRO source statement. Also, you need not define register names before using them in the addressing modes (see Chapter 5).

Register names cannot be redefined; that is, a symbol that you define cannot be one of the register names contained in the following list. You can express the 16 general registers of the VAX processor 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 you use R12 as an argument pointer, the name AP is recommended; if you use R12 as a general register, the name R12 is recommended.
FP	Frame pointer
SP	Stack pointer
PC	Program counter

Note that the symbols IV and DV are also permanent symbols and cannot be redefined. These symbols are used in the register mask to set the integer overflow trap (IV) and the decimal string overflow trap (DV). See Section 3.6.2.2 for an explanation of their uses.

3.3.2 User-Defined Symbols and Macro Names

You can use symbols that you define as labels or you can equate them to a specific value by a direct assignment statement (see Section 3.8). These symbols can also 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, underscores (_), 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).
- Do not use the period (.) in any global symbol name (see Section 3.3.3) because 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. To avoid confusion, give the symbols and macros that you define different names.

3.3.3 Determining Symbol Values

The value of a symbol depends on its use in the program. VAX 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 MACRO searches for a symbol definition in the following order:

1 Previously defined macro names
2 User-defined opcode (see the .OPDEF description in Chapter 6)
3 Permanent symbols (instructions and directives)
4 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. The definition of a global symbol, however, can be referenced from any module in the program.

VAX MACRO treats all symbols that you define as local unless you explicitly declared them to be global by doing any one of the following:

- Use the double colon (::) in defining a label (see Section 2.1).

- Use the double equal sign (= =) in a direct assignment statement (see Section 3.8).
- Use the .GLOBAL, .ENTRY, or .WEAK directive (see Chapter 6).

When your code references a symbol within the module in which it is defined, VAX MACRO considers the reference internal. When your code references a symbol within a module in which it is not defined, VAX MACRO considers the reference external (that is, the symbol is defined externally in another module). You can use the .DISABLE directive to make references to symbols not defined in the current module illegal. In this case, you must use the .EXTERNAL directive to specify that the reference is an external reference. See Chapter 6 for descriptions of the .DISABLE and .EXTERNAL directives.

3.4 Local Labels

Use local labels to identify addresses within a block of source code.

Format

nn$

nn
A decimal integer in the range of 1 to 65535.

Use local labels in the same way as you use the symbol labels that you define, 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 VAX Symbolic Debugger.
- Local labels cannot be used in the .END directive (see Chapter 6).

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 you use the address only within the block. You can use local labels 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, you should not use local labels 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 4.7).

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 6)
- The .ENABLE and .DISABLE directives (see Chapter 6), 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 6.

An example showing the use of local labels follows:

```
RPSUB:  MOVL    AMOUNT,R0        ; Start 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           ; End previous local label
        CLRL    R2               ;   block and start 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 that
ENTR1:  POPR    #^M<R0,R1,R2>    ;   will not be terminated
        ADDL3   R0,R1,R3         ;   by a user-defined label
        BRB     10$              ; Branch to local label that appears
                                 ;   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 by user-defined
NEXT:   CLRL    R4               ;   label terminates local label block
```

3.5 Terms and Expressions

A term can be any 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 previously noted items preceded by a unary operator (see Section 3.6)

VAX MACRO evaluates terms as longword (4-byte) values. If you use an undefined symbol as a term, the linker determines the value of the term. 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 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. In the first case, A and B are multiplied and then C added to the product. In the second case, B and C are added and the sum is multiplied by A. Angle brackets can also be used to apply a unary operator to an entire expression, such as –<A+B>.

If an arithmetic expression is continued on another line, the listing file will not show the continued line. For example:

```
.WORD <DATA1'$^XFF@8+-
      89>
```

You must use /LIST/SHOW=EXPANSION to show the continuation line.

VAX MACRO considers unary operators part of a term and thus, 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), as follows:

- 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, since 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 the following:

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

See Chapter 6 for descriptions of the directives listed in the preceding list.

Expressions used in these directives and in direct assignment statements can contain only 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
                            ;   but contains the symbol LAB3
                            ;   that is defined later in this module
LAB3:   .WORD   TST+LAB+2   ; TST+LAB+2 is an external expression
                            ;   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. You can use unary operators to indicate whether a term or expression is positive or negative. If unary plus or minus is not specified, the default value is assumed to be plus. In addition, unary operators perform radix conversion, textual conversion (including ASCII conversion), and numeric control operations, as described in the following sections. 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 (two'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 one'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 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 is legal in the specified radix. The following are the legal characters for each radix:

Format	Radix Name	Legal Characters
^Bnn	Binary	0 and 1
^Dnn	Decimal	0 to 9
^Onn	Octal	0 to 7
^Xnn	Hexadecimal	0 to 9 and A to 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 default decimal operator is needed only within an expression that has another radix control operator. In the following example, "16" is interpreted as a decimal number because it is preceded by the decimal operator ^D even though the "16" is in an expression prefixed by the octal radix control operator.

```
.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 1 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. Use nonalphanumeric characters to avoid confusion.

Format

^Astring

string
A delimited ASCII string from 1 to 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 Move Word (MOVW) instruction (the data type is a word), the delimited string cannot be more than 2 characters.

For example:

```
.QUAD    ^A%1234/678%     ; Generates 8 bytes of ASCII data
MOVL     #^A/ABCD/,R0     ; Moves characters ABCD
                          ;   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
MOVL     #^A/AB/,R0       ; Moves 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 Push Registers (PUSHR) and Pop Registers (POPR) instructions and the .ENTRY and .MASK directives (see Chapter 6).

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 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 follow.

Register Name	Arithmetic Trap Enable	Bits
R0 to R11		0 to 11
R12 or AP		12
FP		13
SP	IV	14
	DV	15

When the POPR or PUSHR instruction uses the register mask operator, R0 to R11, R12 or AP, FP, and SP can be specified. You cannot specify the PC register name and the IV and DV arithmetic trap-enable specifiers.

When the .ENTRY or .MASK directive uses the register mask operator, you can specify R2 to R11 and the IV and DV arithmetic trap-enable specifiers. However, you cannot specify R0, R1, FP, SP, and PC. IV sets the integer overflow trap, and DV sets the decimal string overflow trap.

The arithmetic trap-enable specifiers are described in Chapter 8.

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 registers 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). The use of the numeric control operators is explained in Section 3.6.3.1 and Section 3.6.3.2.

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 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
MOVF       #3.7,R0         ;   number is the MOVF instruction
```

3.6.3.2 Complement Operator

The complement operator produces the one'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 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 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 you use a 1-byte or 2-byte operand, the result is the low-order bytes of the 4-byte result. VAX 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.

3.7.1 Arithmetic Shift Operator

You use the arithmetic shift operator (@) to perform left and right arithmetic shifts of arithmetic quantities. The first argument is shifted left or right by the number of bit positions that you specify in 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 zero. 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)
A = 4
        .LONG   1@A                 ; Yields 10000 (binary)
        .LONG   ^X1234@-A           ; Yields 123(hex)
        MOVL    #<^B1100000@-5>,R0  ; Yields 11 (binary)
```

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 you use as a label, you can redefine a symbol defined with a direct assignment statement as many times as you want.

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 which defines its value. Spaces preceding 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.

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

Every VAX 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, with the exception of the special form described below, a direct assignment statement 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. This method of setting the location counter is often useful when defining data structures. Data storage areas should not be reserved by explicitly setting the location counter; use the .BLK*x* directives (see Chapter 6).

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 may be relative to the current location counter.

For example:

```
. = .+40          ; Moves location counter forward
```

When a program section that you 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 you use the current location counter 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, you would not normally use the current location counter as a part of the operand specifier.

4 Macro Arguments and String Operators

By using macros, you 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.

The macro directives (described in Chapter 6) provide facilities for performing eight categories of functions. Table 6–2 lists these categories and the directives that fall under them.

By default, macro expansions are not printed in the assembly listing. They are printed only when the .SHOW directive (see description in Chapter 6) or the /SHOW qualifier (described in the *VMS DCL Dictionary*) 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 remainder of this chapter describes macro arguments, created local labels, and the macro string operators.

4.1 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. 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. This strict positional relationship can be overridden by the use of keyword arguments (see Section 4.3).

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 previously:

```
STORE   3,2,1                        ; Macro call
.LONG   3                            ; 3 is first argument
.WORD   1                            ; 1 is third argument
.BYTE   2                            ; 2 is second argument

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
```

4.2 Default Values

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

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 previously:

```
STORE                  ; No arguments supplied
.LONG   12
.WORD   1000
.BYTE   0

STORE   ,5,X           ; Last two arguments supplied
.LONG   12
.WORD   X
.BYTE   5

STORE   1              ; First argument supplied
.LONG   1
.WORD   1000
.BYTE   0
```

4.3 Keyword Arguments

Keyword arguments allow a macro call to specify the arguments in any order. 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 more formal arguments than 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
.BYTE   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.

4.4 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 (<>).

The assembler also interprets any character after an initial circumflex (^) as a delimiter. To pass an angle bracket as part of a string, you 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 (?) 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:

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

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 any delimiters 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.

Strings enclosed by delimiters cannot be continued on a new line.

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 is passed a string argument, you 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 it does not contain any separator characters. The argument is thus delimited because it is a formal argument in the definition of the macro CNTRPT 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. Do not use delimiters around the macro calls in the macro definitions. Each time you use the delimited argument 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 you 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 because the call to CNTRPT2 contains nested delimiters around the string argument.

4.5 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:
```

4.6 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. The value of the symbol can be passed by inserting a backslash (\) before the symbol in the macro call. The assembler 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 previously:

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

4.7 Created Local Labels

Local labels are often very useful in macros. Although you can create a macro definition that specifies local labels within it, these local labels might be duplicated elsewhere in the local label block possibly causing errors. However, the assembler can create local labels in the macro expansion that will not conflict with other local labels. These labels are called created local labels.

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

A created local label is specified by a question mark (?) 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 previously show both created local labels and a user-defined local label:

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

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

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

4.8 Macro String Operators

Following are the three macro string operators:

- %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.

4.8.1 %LENGTH Operator

Format

%LENGTH(string)

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 4.4).

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

EXAMPLES

The macro definition is as follows:

1
```
        .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 nothing
        .ENDM   CHK_SIZE
```

The macro calls and expansions of the macro defined previously are as follows:

2
```
        CHK_SIZE        A                          ; Macro checks if A
        .IF GREATER_EQUAL   1-3                    ;   is between 3 and
        .IF LESS_THAN       6-1                    ;   6 characters long.
                                                   ;   Should be too short.
        .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 nothing
```

```
3        CHK_SIZE        ABC                       ; Macro checks if ABC
         .IF GREATER_EQUAL   3-3                   ;   is between 3 and
         .IF LESS_THAN       6-3                   ;   6 characters long.
                                                   ;   Should be ok.
         .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 nothing
```

4.8.2 %LOCATE Operator

Format

%LOCATE(string1,string2 [,symbol])

Parameters

string1
A substring. The substring can be written either as a macro argument or as a delimited string. The delimiters can be either angle brackets or a character preceded by a circumflex.

string2
The string to be searched for the substring. The string can be written either as a macro argument or as 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 zero (the beginning of the string). The symbol must be an absolute symbol that has been previously defined; the number must be an unsigned decimal number. Expressions and radix operators are not allowed.

DESCRIPTION

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 zero. 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.

EXAMPLES

The macro definition is as follows:

1
```
        .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 nothing
         .ENDM  BIT_NAME
```

The macro calls and expansions of the macro defined previously are as follows:

2
```
        BIT_NAME  ESC            ; Is ESC in list
        .IF EQUAL  12-15         ; If it is not, print error
        .ERROR  ; ESC is an invalid bit name
        .ENDC                    ; If it is, do nothing

        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 nothing
```

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 fifth character position.

4.8.3 %EXTRACT Operator

Format

%EXTRACT(symbol1,symbol2,string)

Parameters

symbol1
A symbol or decimal number that specifies the starting position of the substring to be extracted. The symbol must be an absolute symbol that has been previously defined; the 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 to be extracted. The symbol must be an absolute symbol that has been previously defined; the 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.

DESCRIPTION

The %EXTRACT operator extracts a substring from a string. It returns the substring that begins at the specified position and is of 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 zero.

EXAMPLES

The macro definition is as follows:

1
```
        .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
```

The macro calls and expansions of the macro defined previously are as follows:

2
```
        RESERVE FOOBAR
XX = 6
        .IF EQUAL  XX-6
%MACRO-W-GENWRN, Generated WARNING:  Incorrect format for macro call - FOOBAR

        .MEXIT
```

3
```
        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
```

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

5 VAX MACRO Addressing Modes

This section summarizes the VAX addressing modes and contains examples of VAX MACRO statements that use these addressing modes. Table 5–1 summarizes the addressing modes. (Chapter 8 describes the addressing mode formats in detail.)

The following are the four types of addressing modes:

- General register
- Program counter (PC)
- Index
- Branch

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

5.1 General Register Modes

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

The following are the eight general register modes:

- Register
- Register deferred
- Autoincrement
- Autoincrement deferred
- Autodecrement
- Displacement
- Displacement deferred
- Literal

Table 5–1 Addressing Modes

Type	Addressing Mode	Format	Hex Value	Description	Can Be Indexed?
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^ respectively indicate byte, word, and longword displacement.	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^ respectively indicate, byte, word, and longword displacement.	Yes
	Literal	#literal S^#literal	 0–3	The literal specified is the operand; the literal is stored as a short literal.	No

Key:

Rn—Any general register R0 to R12. Note that the AP, FP, or SP register can be used in place of Rn.
Rx—Any general register R0 to 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 5.3).
dis—An expression specifying a displacement.
address—An expression specifying an address.
literal—An expression, an integer constant, or a floating-point constant.

(continued on next page)

Table 5–1 (Cont.) Addressing Modes

Type	Addressing Mode	Format	Hex Value	Description	Can Be Indexed?
Program counter	Relative	address B^address W^address L^address	 A C E	The address specified is the address of the operand; the address is stored as a displacement from the PC; B^, W^, and L^ respectively indicate byte, word, and longword displacement.	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 the 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 the PC; if the address is defined as an absolute virtual address, the linker stores the address as an absolute value.	Yes

Key:

Rn—Any general register R0 to R12. Note that the AP, FP, or SP register can be used in place of Rn.
Rx—Any general register R0 to 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 5.3).
dis—An expression specifying a displacement.
address—An expression specifying an address.
literal—An expression, an integer constant, or a floating-point constant.

(continued on next page)

Table 5–1 (Cont.) Addressing Modes

Type	Addressing Mode	Format	Hex Value	Description	Can Be Indexed?
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 from the PC; branch mode can only be used with the branch instructions.	No

Key:

Rn—Any general register R0 to R12. Note that the AP, FP, or SP register can be used in place of Rn.
Rx—Any general register R0 to 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 5.3).
dis—An expression specifying a displacement.
address—An expression specifying an address.
literal—An expression, an integer constant, or a floating-point constant.

5.1.1 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 you use the PC in register mode or if you use a large data type that extends the operand into the PC.

Formats

Rn
AP
FP
SP

n
A number in the range 0 to 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
```

5.1.2 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 5.3).

Formats

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

Parameters

n
A number in the range 0 to 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
```

5.1.3 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 5.3), but the index register cannot be the same as the register specified in autoincrement mode.

Formats

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

Parameters

n
A number in the range 0 to 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 2 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.
```

5.1.4 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 5.3), but the index register cannot be the same as the register specified in autoincrement deferred mode.

Formats

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

Parameters

n
A number in the range 0 to 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.
```

5.1.5 Autodecrement Mode

In autodecrement mode, the processor decrements the contents of the register by the size of the operand data type; 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 5.3), but the index register cannot be the same as the register specified in autodecrement mode.

Formats

–(Rn)
–(AP)
–(FP)
–(SP)

Parameters

n
A number in the range 0 to 12.

EXAMPLE

```
CLRO    -(R1)              ; Subtract 8 from R1 and zero
                           ;   the octaword whose address
                           ;   is in R1.

MOVZBL  R3,-(SP)           ; Push the zero-extended low byte
                           ;   of R3 onto the stack as a
                           ;   longword.

CMPB    R1,-(R0)           ; Subtract 1 from R0 and compare
                           ;   low byte of R1 with byte whose
                           ;   address is now in R0.
```

5.1.6 Displacement Mode

In displacement mode, the contents of the register plus the displacement (sign-extended to a longword) produce the address of the operand.

Displacement mode can be used with index mode (see Section 5.3). If used in displacement mode, the index register can be the same as the base register.

Formats

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

Parameters

n
A number in the range 0 to 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 one word (2 bytes).
L^	Displacement requires one 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 one 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 displace-
                               ;   ment 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 zero, and no displacement length is specified, the assembler uses register deferred mode rather than displacement mode.

5.1.7 Displacement Deferred Mode

In displacement deferred mode, the contents of the register plus the displacement (sign-extended to a longword) produce the address of the operand address (a pointer to the operand).

Displacement deferred mode can be used with index mode (see Section 5.3). If used in displacement deferred mode, the index register can be the same as the base register.

Formats

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

Parameters

n
A number in the range 0 to 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 one word (2 bytes).
L^	Displacement requires one 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 one 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 the assembler has 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 + 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 + address of ARRPOINT>
                                   ;   to the address pointed to by
                                   ;   longword whose address is
                                   ;   <RSOFFS + address of ARRPOINT>;
                                   ;   the first displacement is
                                   ;   stored as a byte; the second
                                   ;   displacement is stored as a word.

CLRW    @84(R2)                    ; Clear word pointed to by
                                   ;   <longword at 84 + contents of R2>;
                                   ;   the assembler uses byte
                                   ;   displacement automatically.
```

5.1.8 Literal Mode

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

Formats

#literal
S^#literal

Parameters

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 0 to 63 and floating-point constants must be one of the 64 values listed in Table 5–2 and Table 5–3. Floating-point short literals are stored with a 3-bit exponent and a 3-bit fraction. Table 5–2 and Table 5–3 also show the value of the exponent and the fraction for each literal. See Section 8.6.8 for information on the format of short literals.

Table 5–2 Floating-Point Literals Expressed as Decimal Numbers

Exponent	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

Table 5–3 Floating-Point Literals Expressed as Rational Numbers

Exponent	0	1	2	3	4	5	6	7
0	1/2	9/16	5/8	11/16	3/4	13/16	7/8	15/16
1	1	1-1/8	1-1/4	1-3/8	1-1/2	1-5/8	1-3/4	1-7/8
2	2	2-1/4	2-1/2	2-3/4	3	3-1/4	3-1/2	3-3/4
3	4	4-1/2	5	5-1/2	6	6-1/2	7	7-1/2
4	8	9	10	11	12	13	14	15
5	16	18	20	22	24	26	28	30
6	32	36	40	44	48	52	56	60
7	64	72	80	88	96	104	112	120

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 you use the #literal format, the assembler chooses whether to use literal mode or immediate mode (see Section 5.2.4). The assembler uses immediate mode if any of the following conditions is satisfied:

- 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.

5.2 Program Counter Modes

The program counter (PC) modes use the PC for a general register. Following are the five program counter modes:

- Relative
- Relative deferred
- Absolute
- Immediate
- General

In Section 8.7, Table 8–6 is a summary of PC addressing.

5.2.1 Relative Mode

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

Relative mode can be used with index mode (see Section 5.3).

Format

address

Parameters

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 one word (2 bytes).
L^	Displacement requires one 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 6). 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 was
                           ;   previously defined in this
                           ;   section

CMPL    W^<DATA+4>,R10     ; Compare R10 with longword at
                           ;   address DATA+4; CMPL
                           ;   uses a word displacement
```

5.2.2 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 the PC.

Relative deferred mode can be used with index mode (see Section 5.3).

Format

@address

Parameters

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 one word (2 bytes).
L^	Displacement requires one 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 6). 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
```

5.2.3 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 the PC).

Absolute mode can be used with index mode (see Section 5.3).

Format

@#address

Parameters

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
```

5.2.4 Immediate Mode

In immediate mode, the literal specified is the operand.

Formats

#literal
I^#literal

Parameters

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 cannot 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
```

```
MOVG        #0.2,R6        ; The value 0.2 is converted
                           ;   to its G_FLOATING representation

MOVG        #PI,R6         ; The value contained in PI is
                           ;   moved to R6; no conversion is
                           ;   performed
```

Notes

1 When you use the #literal format, the assembler chooses whether to use literal mode (Section 5.1.8) or immediate mode. If the literal is an integer from 0 to 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.

3 You can specify floating-point numbers two ways: as a numeric value or as a symbol name. The assembler handles these values in different ways, as follows:

- Numeric values are converted to the appropriate internal floating-point representation.
- Symbols are not converted. The assembler assumes that the values have already been converted to internal floating-point representation.

Once the assembler obtains the value, it tries to convert the internal representation of the value to a short floating literal. If conversion fails, the assembler uses immediate mode; if conversion succeeds, the assembler uses short floating literal mode.

5.2.5 General Mode

In general mode, the address you specify 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. You should use general mode to write position-independent code when you do not know whether the address is relocatable or absolute. A general addressing mode operand requires 5 bytes of storage.

You can use general mode with index mode (see Section 5.3).

Format

G^address

Parameters

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 general mode is
                            ;   converted to absolute
                            ;   mode; if it is defined as
                            ;   relocatable, then general mode is
                            ;   converted to relative mode
CALLS   #5,G^SYS$SERVICE    ; Calls procedure SYS$SERVICE
                            ;   with 5 arguments on stack
```

5.3 Index Mode

Index mode is a general register mode that can be used only in combination with another mode (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. To get the operand address, the assembler multiplies the contents of the index register by the number of bytes of the operand data type, then adds the result to the base 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]

Parameters

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

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

Table 5–4 lists the formats of index mode addressing.

EXAMPLE

```
;
; 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 5–4 Index Mode Addressing

Mode	Format
Register Deferred Index[1,2]	(Rn)[Rx]
Autoincrement Index[1,2]	(Rn)+[Rx]
Autoincrement Deferred Index[1,2]	@(Rn)+[Rx]
Autodecrement Index[1,2]	–(Rn)[Rx]
Displacement Index[1,2,3]	dis(Rn)[Rx]
Displacement Deferred Index[1,2,3]	@dis(Rn)[Rx]
Relative Index[2]	address[Rx]
Relative Deferred Index[2]	@address[Rx]
Absolute Index[2]	@#address[Rx]
General Index[2]	G^address[Rx]

[1]Rn—Any general register R0 to R12 or the AP, FP, or SP register.

[2]Rx—Any general register R0 to 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.

[3]dis—An expression specifying a displacement.

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.

5.4 Branch Mode

In branch mode, the address is stored as an implied displacement from the PC. This mode can be used only 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 32,767 bytes forward and 32,768 bytes backward. The displacement is relative to the updated PC, the byte past the byte or word where the displacement is stored. See Chapter 9 for more information on the branch instructions.

Format

address

Parameters

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
```