6 VAX MACRO Assembler Directives

The general assembler directives provide facilities for performing 11 types of functions. Table 6–1 lists these types of functions and their directives.

The macro directives provide facilities for performing eight categories of functions. Table 6–2 lists these categories and their associated directives. Chapter 4 describes macro arguments and string operators.

The remainder of this chapter describes both the general assembler directives and the macro directives, showing their formats and giving examples of their use. For ease of reference, the directives are presented in alphabetical order. Appendix C contains a summary of all assembler directives.

Table 6–1 Summary of General Assembler Directives

Category	Directives[1]
Listing control directives	.SHOW (.LIST)
	.NOSHOW(.NLIST)
	.TITLE
	.SUBTITLE (.SBTTL)
	.IDENT
	.PAGE
Message display directives	.PRINT
	.WARN
	.ERROR
Assembler option directives	.ENABLE (.ENABL)
	.DISABLE(.DSABL)
	.DEFAULT

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

(continued on next page)

Table 6–1 (Cont.) **Summary of General Assembler Directives**

Category	Directives[1]
Data storage directives	.BYTE .WORD .LONG .ADDRESS .QUAD .OCTA .PACKED .ASCII .ASCIC .ASCID .ASCIZ .F_FLOATING (.FLOAT) .D_FLOATING (.DOUBLE) .G_FLOATING .H_FLOATING .SIGNED_BYTE .SIGNED_WORD
Location control directives	.ALIGN .EVEN .ODD .BLKA .BLKB .BLKD .BLKF .BLKG .BLKH .BLKL .BLKO .BLKQ .BLKW .END
Program sectioning directives	.PSECT .SAVE_PSECT (.SAVE) .RESTORE_PSECT (.RESTORE)
Symbol control directives	.GLOBAL (.GLOBL) .EXTERNAL (.EXTRN) .DEBUG .WEAK
Routine entry point definition directives	.ENTRY .TRANSFER .MASK

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

(continued on next page)

Table 6–1 (Cont.) Summary of General Assembler Directives

Category	Directives[1]
Conditional and subconditional assembly block directives	.IF .ENDC .IF_FALSE (.IFF) .IF_TRUE (.IFT) .IF_TRUE_FALSE (.IFTF) .IIF
Cross-reference directives	.CROSS .NOCROSS
Instruction generation directives	.OPDEF .REF1 .REF2 .REF4 .REF8 .REF16
Linker option record directive	.LINK

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

Table 6–2 Summary of Macro Directives

Category	directives[1]
Macro definition directives	.MACRO .ENDM
Macro library directives	.LIBRARY .MCALL
Macro deletion directive	.MDELETE
Macro exit directive	.MEXIT
Argument attribute directives	.NARG .NCHR .NTYPE
Indefinite repeat block directives	.IRP .IRPC
Repeat block directives	.REPEAT (.REPT)
End range directive	.ENDR

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

.ADDRESS

Address storage directive

FORMAT

.ADDRESS *address-list*

PARAMETER

address-list
A list of symbols or expressions, separated by commas (,), which VAX MACRO interprets as addresses. Repetition factors are not allowed.

DESCRIPTION

.ADDRESS stores successive longwords containing addresses in the object module. Digital recommends that you use .ADDRESS rather than .LONG for storing address data to provide additional information to the linker. In shareable images, addresses that you specify with .ADDRESS produce position-independent code.

EXAMPLE

```
TABLE:  .ADDRESS  LAB_4, LAB_3, ROUTTERM        ; Reference table
```

.ALIGN

Location counter alignment directive

FORMAT

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

PARAMETERS

integer

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

keyword

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

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

expression

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

DESCRIPTION

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

Notes

1 The alignment that you specify in .ALIGN cannot exceed the alignment of the program section in which the alignment is attempted (see the description of .PSECT). For example, if you are using the default program section alignment (BYTE) and you specify .ALIGN with a word or larger alignment, the assembler displays an error message. fills the bytes skipped by the location counter (if any) with the value of that expression. Otherwise, the assembler fills the bytes with zeros.

2 Although most instructions can use byte alignment of data, execution speed is improved by the following alignments:

Data Length	Alignment
Word	Word
Longword	Longword
Quadword	Quadword

EXAMPLE

```
.ALIGN   BYTE,0        ; Byte alignment--fill with null
.ALIGN   WORD          ; Word alignment
.ALIGN   3,^A/ /       ; Quad alignment--fill with blanks
.ALIGN   PAGE          ; Page alignment
```

.ASCI*x*

ASCII character storage directives

DESCRIPTION

VAX MACRO has the following four ASCII character storage directives:

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

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

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

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

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

ASCII strings can be continued over several lines. Use the hyphen (-) as the line continuation character and delimit the string on each line at both ends. Note that you can use a different pair of delimiters for each line. For example:

```
CR=13
LF=10

        .ASCII        /ABC DEFG/
        .ASCIZ        @Any character can be a delimiter@
        .ASCIC        ? lowercase is not converted to UPPER?
        .ASCII        ? this is a test!?<CR><KEY>(LF\TEXT)!Isn't it?!
        .ASCII        \ Angle Brackets <are part <of> this> string \
        .ASCII        / This string is continued / -
                      \ on the next line \
        .ASCII        <CR><KEY>(LF\TEXT)! this string includes an expression! -
                      <128+CR>? whose value is a 13 plus 128?
```

.ASCIC

Counted ASCII string storage directive

FORMAT **.ASCIC** *string*

PARAMETER

string
A delimited ASCII string.

DESCRIPTION

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

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

EXAMPLE

```
CR=13                                 ; Direct assignment statement
                                      ;   defines CR
       .ASCIC       #HELLO#<CR>       ; This counted ASCII string
                                      ;   is equivalent to the
       .BYTE        6                 ;   count followed by
       .ASCII       #HELLO#<CR>       ;   the ASCII string
```

.ASCID

String-descriptor ASCII string storage directive

FORMAT

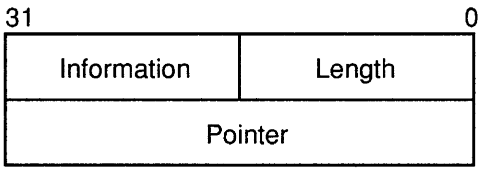
.ASCID *string*

PARAMETER

string
A delimited ASCII string.

DESCRIPTION

.ASCID performs the same function as ASCII, except that .ASCID inserts a string descriptor before the string data. The string descriptor has the following format:



ZK–0370–GE

Parameters

length
The length of the string (2 bytes).

information
Descriptor information (2 bytes) is always set to 010E.

pointer
Position-independent pointer to the string (4 bytes).

String descriptors are used in calling procedures (see the *VMS RTL String Manipulation (STR$) Manual*).

EXAMPLE

```
DESCR1:  .ASCID   /ARGUMENT FOR CALL/         ; String descriptor
DESCR2:  .ASCID   /SECOND ARGUMENT/           ; Another string
                                              ;   descriptor
           .
           .
           .
         PUSHAL   DESCR1                      ; Put address of descriptors
         PUSHAL   DESCR2                      ;   on the stack
         CALLS    #2,STRNG_PROC               ; Call procedure
```

.ASCII

ASCII string storage directive

FORMAT

.ASCII *string*

PARAMETER

string
A delimited ASCII string.

DESCRIPTION

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

EXAMPLE

```
CR=13                                 ; Assignment statements
LF=10                                 ;   define CR and LF

       .ASCII   "DATE: 17-NOV-1988"   ; Delimiter is "
       .ASCII   /EOF/<CR><LF>       ; Delimiter is /
```

.ASCIZ

Zero-terminated ASCII string storage directive

FORMAT

.ASCIZ *string*

PARAMETER

string
A delimited ASCII string.

DESCRIPTION

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

EXAMPLE

```
FF=12                                    ; Define FF
        .ASCIZ   /ABCDEF/                ; 6 characters in string,
                                         ;   7 bytes of data
        .ASCIZ   /A/<KEY>(FF\TEXT)/B/              ; 3 characters in strings,
                                         ;   4 bytes of data
```

.BLK*x*

Block storage allocation directives

FORMAT

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

PARAMETER

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

DESCRIPTION

VAX MACRO has the following 10 block storage directives.

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

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

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

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

EXAMPLE

```
.BLKB   15              ; Space for 15 bytes
.BLKO   3               ; Space for 3 octawords (48 bytes)
.BLKL   1               ; Space for 1 longword (4 bytes)
.BLKF   <3*4>           ; Space for 12 single-precision
                        ;   floating-point values (48 bytes)
```

.BYTE

Byte storage directive

FORMAT

.BYTE *expression-list*

PARAMETER

expression-list

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

Optionally, each expression can be followed by a repetition factor delimited by square brackets ([ ]). An expression followed by a repetition factor has the following format:

expression1 [expression2]

expression1

An expression that specifies the value to be stored.

[expression2]

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

DESCRIPTION

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

Notes

1. The assembler displays an error message if the high-order 3 bytes of the longword expression have a value other than 0 or ^XFFFFFF.
2. At link time, a relocatable expression can result in a value that exceeds 1 byte in length. In this case, the linker issues a truncation diagnostic message for the object module in question. For example:

```
A:      .BYTE   A       ; Relocatable value 'A' will
                        ;   cause linker truncation
                        ;   diagnostic if the statement
                        ;   has a virtual address of 256
                        ;   or above
```

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

EXAMPLE

```
.BYTE    <1024-1000>*2           ; Stores a value of 48
.BYTE    ^XA,FIF,10,65-<21*3>    ; Stores 4 bytes of data
.BYTE    0                       ; Stores 1 byte of data
.BYTE    X,X+3[5*4],Z            ; Stores 22 bytes of data
```

.CROSS
.NOCROSS

Cross-reference directives

FORMAT **.CROSS** *[symbol-list]*
.NOCROSS *[symbol-list]*

PARAMETER ***symbol-list***
A list of legal symbol names separated by commas (,).

DESCRIPTION When you specify the /CROSS_REFERENCE qualifier in the MACRO command, VAX MACRO produces a cross-reference listing. The .CROSS and .NOCROSS directives control which symbols are included in the cross-reference listing. The .CROSS and .NOCROSS directives have an effect only if /CROSS_REFERENCE was specified in the MACRO command (see the *VMS DCL Dictionary*).

By default, the cross-reference listing includes the definition and all the references to every symbol in the module.

You can disable the cross-reference listing for all symbols or for a specified list of symbols by using .NOCROSS. Using .NOCROSS without a symbol list disables the cross-reference listing of all symbols. Any symbol definition or reference that appears in the code after .NOCROSS used without a symbol list and before the next .CROSS used without a symbol list is excluded from the cross-reference listing. You reenable the cross-reference listing by using .CROSS without a symbol list.

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

Notes

1. The .CROSS directive without a symbol list will not reenable the cross-reference listing of a symbol specified in .NOCROSS with a symbol list.
2. If the cross-reference listing of all symbols is disabled, .CROSS with a symbol list will have no effect until the cross-reference listing is reenabled by .CROSS without a symbol list.

EXAMPLES

1
```
          .NOCROSS                ; Stop cross-reference
LAB1:     MOVL       LOC1, LOC2   ; Copy data
          .CROSS                  ; Reenable cross-reference
```

In this example, the definition of LAB1 and the references to LOC1 and LOC2 are not included in the cross-reference listing.

2
```
          .NOCROSS  LOC1        ; Do not cross-reference LOC1
LAB2:     MOVL      LOC1,LOC2   ; Copy data
          .CROSS    LOC1        ; Reenable cross-reference
                                ;   of LOC1
```

In this example, the definition of LAB2 and the reference to LOC2 are included in the cross-reference, but the reference to LOC1 is not included in the cross-reference.

.DEBUG

Debug symbol attribute directive

FORMAT

.DEBUG *symbol-list*

PARAMETER

symbol-list
A list of legal symbols separated by commas (,).

DESCRIPTION

.DEBUG specifies that the symbols in the list are made known to the VAX Symbolic Debugger. During an interactive debugging session, you can use these symbols to refer to memory locations or to examine the values assigned to the symbols.

Note

The assembler adds the symbols in the symbol list to the symbol table in the object module. You need not specify global symbols in the .DEBUG directive because global symbols are automatically put in the object module's symbol table. (See the description of .ENABLE for a discussion of how to make information about local symbols available to the debugger.)

EXAMPLE

```
.DEBUG   INPUT,OUTPUT,-      ; Make these symbols known
         LAB_30,LAB_40       ;   to the debugger
```

.DEFAULT

Default control directive

FORMAT **.DEFAULT** *DISPLACEMENT, keyword*

PARAMETER

keyword
One of three keywords—BYTE, WORD, or LONG—indicating the default displacement length.

DESCRIPTION

.DEFAULT determines the default displacement length for the relative and relative deferred addressing modes (see Section 5.2.1 and Section 5.2.2).

Notes

1 The .DEFAULT directive has no effect on the default displacement for displacement and displacement deferred addressing modes (see Section 5.1.6
and Section 5.1.7).

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

EXAMPLE

```
.DEFAULT  DISPLACEMENT,WORD    ; WORD is default
MOVL      LABEL,R1             ; Assembler uses word
                               ;   displacement unless
                               ;   label has been defined
.DEFAULT  DISPLACEMENT,LONG    ; LONG is default
INCB    @COUNTS+4              ; Assembler uses longword
                               ;   displacement unless
                               ;   COUNTS has been defined
```

.D_FLOATING
.DOUBLE

Floating-point storage directive

FORMAT **.D_FLOATING** *literal-list*
.DOUBLE *literal-list*

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

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

Notes

1 Double-precision floating-point numbers are always rounded. They are not affected by .ENABLE TRUNCATION.

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

EXAMPLE

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

.DISABLE

Function control directive

FORMAT

.DISABLE *argument-list*

PARAMETER

argument-list
One or more of the symbolic arguments listed in Table 6–3 in the description of .ENABLE. You can use either the long or the short form of the symbolic arguments. If you specify multiple arguments, separate them by commas (,), spaces, or tabs.

DESCRIPTION

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

Note

The alternate form of .DISABLE is .DSABL.

.ENABLE

Function control directive

FORMAT **.ENABLE** *argument-list*

PARAMETER ***argument-list***

One or more of the symbolic arguments listed in Table 6–3. You can use either the long form or the short form of the symbolic arguments.

If you specify multiple arguments, separate them with commas (,), spaces, or tabs.

Table 6–3 .ENABLE and .DISABLE Symbolic Arguments

Long Form	Short Form	Default Condition	Function
ABSOLUTE	AMA	Disabled	When ABSOLUTE is enabled, all the PC relative addressing modes are assembled as absolute addressing modes.
DEBUG	DBG	Disabled	When DEBUG is enabled, all local symbols are included in the object module's symbol table for use by the debugger.
GLOBAL	GBL	Enabled	When GLOBAL is enabled, all undefined symbols are considered external symbols. When GLOBAL is disabled, any undefined symbol that is not listed in an .EXTERNAL directive causes an assembly error.
LOCAL_ BLOCK	LSB	Disabled	When LOCAL_BLOCK is enabled, the current local label block is ended and a new one is started. When LOCAL_BLOCK is disabled, the current local label block is ended. See Section 3.4 for a complete description of local label blocks.

(continued on next page)

Table 6–3 (Cont.) .ENABLE and .DISABLE Symbolic Arguments

Long Form	Short Form	Default Condition	Function
SUPPRESSION	SUP	Disabled	When SUPPRESSION is enabled, all symbols that are defined but not referred to are not listed in the symbol table. When SUPPRESSION is disabled, all symbols that are defined are listed in the symbol table.
TRACEBACK	TBK	Enabled	When TRACEBACK is enabled, the program section names and lengths, module names, and routine names are included in the object module for use by the debugger. When TRACEBACK is disabled, VAX MACRO excludes this information and, in addition, does not make any local symbol information available to the debugger.
TRUNCATION	FPT	Disabled	When TRUNCATION is enabled, single-precision, floating-point numbers are truncated. When TRUNCATION is disabled, single-precision floating-point numbers are rounded. D_floating, G_floating, and H_floating numbers are not affected by .ENABLE TRUNCATION; they are always rounded.
VECTOR		Disabled	When VECTOR is enabled, the assembler accepts and correctly handles vector code. If vector assembly is not enabled, vector code produces assembly errors.

DESCRIPTION

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

- Creating local label blocks
- Making all local symbols available to the debugger and enabling the traceback feature
- Specifying that undefined symbol references are external references
- Truncating or rounding single-precision floating-point numbers

- Suppressing the listing of symbols that are defined but not referenced
- Specifying that all the PC references are absolute, not relative

Note

The alternate form of .ENABLE is .ENABL.

EXAMPLE

```
.ENABLE ABSOLUTE, GLOBAL        ; Assemble relative address mode
                                ;   as absolute address mode, and consider
                                ;   undefined references as global

.DISABLE TRUNCATION,TRACEBACK   ; Round floating-point numbers, and
                                ;   omit debugging information from
                                ;   the object module
```

.END

Assembly termination directive

FORMAT

.END *[symbol]*

PARAMETER

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

DESCRIPTION

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

Notes

1. The transfer address must be in a program section that has the EXE attribute (see the description of .PSECT).
2. When an executable image consisting of several object modules is linked, only one object module should be terminated by an .END directive that specifies a transfer address. All other object modules should be terminated by .END directives that do not specify a transfer address. If an executable image contains either no transfer address or more than one transfer address, the linker displays an error message.
3. If the source program contains an unterminated conditional code block when the .END directive is specified, the assembler displays an error message.

EXAMPLE

```
.ENTRY  START,0                 ; Entry mask
  .
  .                             ; Main program
  .
.END    START                   ; Transfer address
```

.ENDC

End conditional directive

FORMAT **.ENDC**

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

.ENDM

End definition directive

FORMAT

.ENDM *[macro-name]*

PARAMETERS

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

DESCRIPTION

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

Note

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

.ENDR

End range directive

FORMAT

.ENDR

DESCRIPTION

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

.ENTRY

Entry directive

FORMAT

.ENTRY *symbol,expression*

PARAMETERS

symbol
The symbolic name for the entry point.

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

DESCRIPTION

.ENTRY defines a symbolic name for an entry point and stores a register save mask (2 bytes) at that location. The symbol is defined as a global symbol with a value equal to the value of the location counter at the .ENTRY directive. You can use the entry point as the transfer address of the program. Use the register save mask to determine which registers are saved before the procedure is called. These saved registers are automatically restored when the procedure returns control to the calling program. See the description of the procedure call instructions in Chapter 9.

Notes

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

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

3 Digital recommends that you use .ENTRY to define all callable entry points including the transfer address of the program. Although the following construct also defines an entry point, Digital discourages its use:

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

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

4 You should use .ENTRY only for procedures that are called by the CALLS or CALLG instruction. A routine that is entered by the BSB or JSB instruction should not use .ENTRY because these instructions do not expect a register save mask. Begin these routines using the following format:

```
symbol:: first instruction
```

The first instruction of the routine immediately follows the symbol.

EXAMPLE

```
.ENTRY   CALC,^M<R2,R3,R7>        ; Procedure starts here.
                                  ; Registers R2, R3, and R7
                                  ;   are preserved by CALL
                                  ;   and RET instructions
```

.ERROR

Error directive

FORMAT

.ERROR *[expression] ;comment*

PARAMETERS

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

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

DESCRIPTION

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

Notes

1. .ERROR, .WARN, and .PRINT are message display directives. Use them to display information indicating that a macro call contains an error or an illegal set of conditions.
2. When the assembly is finished, the assembler displays the total number of errors, warnings, information messages, and the sequence numbers of the lines causing the errors or warnings.
3. If .ERROR is included in a macro library, end the comment with a semicolon (;). Otherwise, the librarian will strip the comment from the directive and it will not be displayed when the macro is called.
4. The line containing the .ERROR directive is not included in the listing file.
5. If the expression has a value of zero, it is not displayed in the error message.

EXAMPLE

```
.IF DEFINED      LONG_MESS
.IF GREATER      1000-WORK_AREA
.ERROR 25                         ; Need larger WORK_AREA;
.ENDC
.ENDC
```

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

```
%MACRO-E-GENERR, Generated ERROR: 25 Need larger WORK_AREA
```

.EVEN

Even location counter alignment directive

FORMAT **.EVEN**

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

.EXTERNAL

External symbol attribute directive

FORMAT

.EXTERNAL *symbol-list*

PARAMETER

symbol-list
A list of legal symbols, separated by commas (,).

DESCRIPTION

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

Notes

1 If the GLOBAL argument is enabled (see Table 6–3), all unresolved references will be marked as global and external. If GLOBAL is enabled, you need not specify .EXTERNAL. If GLOBAL is disabled, you must explicitly specify .EXTERNAL to declare any symbols that are defined externally but are referred to in the current module.

2 If GLOBAL is disabled and the assembler finds symbols that are neither defined in the current module nor listed in a .EXTERNAL directive, the assembler displays an error message.

3 Note that if your program does not reference, in a relocatable program section, symbols that are declared in the absolute program section (ABS), the unreferenced symbols are filtered out by the assembler and will not be included in the object file. This filtering out will occur even if the symbols are declared global or external.

If you want to be sure that a symbol will be included even if it is not referenced, declare it in a relocatable program section. If you want to make sure that a symbol you define in an absolute program section is included, reference it in a relocatable program section.

4 The alternate form of .EXTERNAL is .EXTRN.

EXAMPLE

```
.EXTERNAL      SIN,TAN,COS      ; These symbols are defined in
.EXTERNAL      SINH,COSH,TANH   ;   externally assembled modules
```

.F_FLOATING
.FLOAT

Floating-point storage directive

FORMAT

.F_FLOATING *literal-list*
.FLOAT *literal-list*

PARAMETER

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

DESCRIPTION

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

Notes

1. See the description of .ENABLE for information on specifying floating-point rounding or truncation.
2. The floating-point constants in the literal list must not be preceded by the floating-point unary operator (^F).

EXAMPLE

```
.F_FLOATING  134.5782,74218.34E20      ; Constant list
.F_FLOATING  134.2,0.1342E3,1342E-1    ; These all generate 134.2
.F_FLOATING  -0.75,1E38,-1.0E-37       ; Constant list
.FLOAT       0,25,50
```

.G_FLOATING

G_floating-point storage directive

FORMAT

.G_FLOATING *literal-list*

PARAMETERS

literal-list

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

DESCRIPTION

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

Notes

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

EXAMPLE

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

.GLOBAL

Global symbol attribute directive

FORMAT **.GLOBAL** *symbol-list*

PARAMETER ***symbol-list***
A list of legal symbol names, separated by commas (,).

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

Notes

1 .GLOBAL is provided for MACRO–11 compatibility only. Digital recommends that global definitions be specified by a double colon (::) or double equal sign (==) (see Section 2.1 and Section 3.8) and that external references be specified by .EXTERNAL when necessary.

2 The alternate form of .GLOBAL is .GLOBL.

EXAMPLE

```
.GLOBAL LAB_40,LAB_30           ; Make these symbol names
                                ;   globally known
.GLOBAL UKN_13                  ;   to all linked modules
```

.H_FLOATING

H_floating-point storage directive

FORMAT

.H_FLOATING *literal-list*

PARAMETER

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

DESCRIPTION

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

Notes

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

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

EXAMPLE

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

.IDENT

Identification directive

FORMAT

.IDENT *string*

PARAMETER

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

DESCRIPTION

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

Notes

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

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

EXAMPLE

```
.IDENT  /3-47/                ; Version and edit numbers
```

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

.IF

Conditional assembly block directives

FORMAT

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

PARAMETERS

condition
A specified condition that must be met if the block is to be included in the assembly. The condition must be separated from the argument by a comma (,), space, or tab. Table 6–4 lists the conditions that can be tested by the conditional assembly directives.

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

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

Table 6–4 Condition Tests for Conditional Assembly Directives

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

[1]The BLANK, NOT_BLANK, IDENTICAL, and DIFFERENT conditions are only useful in macro definitions.

DESCRIPTION

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

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

Notes

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

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

3 Lowercase string arguments are converted to uppercase before being compared, unless the string is surrounded by delimiters. For information on string arguments and delimiters, see Chapter 4.

4 The assembler displays an error message if .IF specifies any of the following: a condition test other than those in Table 6–4, an illegal argument, or a null argument specified in an .IF directive.

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

EXAMPLES

1 An example of a conditional assembly directive is:

```
    .IF EQUAL  ALPHA+1         ; Assemble block if ALPHA+1=0. Do
      .                        ;   not assemble if ALPHA+1 not=0
      .
      .
    .ENDC
```

2 Nested conditional directives take the form:

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

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

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

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

.IF_x

Subconditional assembly block directives

FORMAT

.IF_FALSE
.IF_TRUE
.IF_TRUE_FALSE

DESCRIPTION

VAX MACRO has the following three subconditional assembly block directives:

Directive	Function
.IF_FALSE	If the condition of the assembly block tests false, the program includes the source code following the .IF_FALSE directive and continuing up to the next subconditional directive or to the end of the conditional assembly block.
.IF_TRUE	If the condition of the assembly block tests true, the program includes the source code following the .IF_TRUE directive and continuing up to the next subconditional directive or to the end of the conditional assembly block.
.IF_TRUE_FALSE	Regardless of whether the condition of the assembly block tests true or false, the source code following the .IF TRUE_ FALSE directive (and continuing up to the next subconditional directive or to the end of the assembly block) is always included.

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

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

Notes

1. If a subconditional directive appears outside a conditional assembly block, the assembler displays an error message.
2. The alternate forms of .IF_FALSE, .IF_TRUE, and .IF_TRUE_FALSE are .IFF, .IFT, and .IFTF.

EXAMPLES

1 Assume that symbol SYM is defined:

```
.IF DEFINED   SYM                 ; Tests TRUE since SYM is defined.
  .                               ;   Assembles the following code.
  .
  .
.IF_FALSE                         ; Tests FALSE since previous
  .                               ;   .IF was TRUE.  Does not
  .                               ;   assemble the following code.
  .
.IF_TRUE                          ; Tests TRUE since SYM is defined.
  .                               ;   Assembles the following code.
  .
  .
.IF_TRUE_FALSE                    ; Assembles following code
  .                               ;   unconditionally.
  .
  .
.IF_TRUE                          ; Tests TRUE since SYM is defined.
  .                               ;   Assembles remainder of
  .                               ;   conditional assembly block.
  .
.ENDC
```

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

```
.IF DEFINED  X                    ; Tests TRUE since X is defined.
.IF DEFINED  Y                    ; Tests FALSE since Y is not defined.
.IF_FALSE                         ; Tests TRUE since Y is not defined.
  .                               ;   Assembles the following code.
  .
  .
.IF_TRUE                          ; Tests FALSE since Y is not defined.
  .                               ;   Does not assemble the following
  .                               ;   code.
  .
.ENDC
.ENDC
```

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

```
.IF DEFINED  A                    ; Tests TRUE since A is defined.
  .                               ;   Assembles the following code.
  .
  .
.IF_FALSE                         ; Tests FALSE since A is defined.
  .                               ;   Does not assemble the following
  .                               ;   code.
  .
.IF NOT_DEFINED B                 ; Nested conditional directive
  .                               ;   is not evaluated.
  .
  .
.ENDC
.ENDC
```

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

```
.IF DEFINED  X                 ; Tests FALSE since X is not defined.
  .                            ;   Does not assemble the following
  .                            ;   code.
  .
.IF DEFINED  Y                 ; Nested conditional directive
  .                            ;   is not evaluated.
  .
  .
.IF_FALSE                      ; Nested subconditional
  .                            ;   directive is not evaluated.
  .
  .
.IF_TRUE                       ; Nested subconditional
  .                            ;   directive is not evaluated.
  .
  .
.ENDC
.ENDC
```

.IIF

Immediate conditional assembly block directive

FORMAT

.IIF *condition [,]argument(s), statement*

PARAMETERS

condition

One of the legal condition tests defined for conditional assembly blocks in Table 6–4 (see the description of .IF). The condition must be separated from the arguments by a comma (,), space, or tab. If the first argument can be a blank, the condition must be separated from the arguments with a comma.

argument(s)

An expression or symbolic argument (described in Table 6–4) associated with the immediate conditional assembly block directive. If the argument is an expression, it cannot contain any undefined symbols and must be an absolute expression (see Section 3.5). The arguments must be separated from the statement by a comma.

statement

The statement to be assembled if the condition is satisfied.

DESCRIPTION

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

Note

The assembler displays an error message if .IIF specifies a condition test other than those listed in Table 6–4, an illegal argument, or a null argument.

EXAMPLE

```
.IIF DEFINED EXAM, BEQL ALPHA
```

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

```
BEQL    ALPHA
```

.IRP

Indefinite repeat argument directive

FORMAT

.IRP *symbol,<argument list>*

.
.
.

range

.
.
.

.ENDR

PARAMETERS

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

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

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

DESCRIPTION

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

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

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

EXAMPLE

The macro definition is as follows:

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

The macro call and expansion of the macro defined previously is as follows:

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

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

.IRPC

Indefinite repeat character directive

FORMAT

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

PARAMETERS

symbol

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

<STRING>

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

range

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

DESCRIPTION

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

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

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

EXAMPLE

The macro definition is as follows:

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

The macro call and expansion of the macro defined previously is as follows:

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

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

.LIBRARY

Macro library directive

FORMAT **.LIBRARY** *macro-library-name*

PARAMETERS

macro-library-name
A delimited string that is the file specification of a macro library.

DESCRIPTION

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

If you omit any information from the macro-library-name argument, default values are assumed. The device defaults to your current default disk; the directory defaults to your current default directory; the file type defaults to MLB.

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

EXAMPLE

```
.LIBRARY    /DISK:[TEST]USERM/       ; DISK:[TEST]USERM.MLB
.LIBRARY    ?DISK:SYSDEF.MLB?        ; DISK:SYSDEF.MLB
.LIBRARY    \CURRENT.MLB\            ; Uses default disk and directory
```

.LINK

Linker option record directive

FORMAT **.LINK** *"file-spec" [/qualifier[=(module-name[,...])],...]*

PARAMETERS

file-spec[,...]
A delimited string that specifies one or more input files. The delimiters can be any matching pair of printable characters except the space, tab, equal sign (=), semicolon (;), or left angle bracket (<). The character that you use as the delimiter cannot appear in the string itself. Although you can use alphanumeric characters as delimiters, use nonalphanumeric characters to avoid confusion.

The input files can be object modules to be linked, or shareable images to be included in the output image. Input files can also be libraries containing external references or specific modules for inclusion in the output image. The linker will search the libraries for the external references. If you specify multiple input files, separate the file specifications with commas (,).

If you do not specify a file type in an input file specification, the linker supplies default file types, based on the nature of the file. All object modules are assumed to have file types of OBJ.

Note that the input file specifications must be correct at *link* time. Make your references explicit, so that if the object module created by VAX MACRO is linked in a directory other than the one in which it was created, the linker will still be able to find the files referenced in the .LINK directive.

No wildcard characters are allowed in the file specification.

FILE QUALIFIERS

/INCLUDE=(module-name[,...])
Indicates that the associated input file is an object library or shareable image library, and that only the module names specified are to be unconditionally included as input to the linker.

At least one module name must be specified. If you specify more than one module name, separate the names with commas (,) and enclose the list in parentheses.

No wildcard characters are allowed in the module name specifications. Module names may not be longer than 31 characters, the maximum length of a VAX MACRO symbol.

/LIBRARY
Indicates that the associated input file is a library to be searched for modules to resolve any undefined symbols in the input files.

If the associated input file specification does not include a file type, the linker assumes the default file type of OLB. You can use both /INCLUDE and /LIBRARY to qualify a file specification. If you specify both /INCLUDE and /LIBRARY, the library is subsequently searched for unresolved references. In this case, the explicit inclusion of modules occurs first; then the linker searches the library for unresolved references.

/SELECTIVE_SEARCH

Directs the linker to add to its symbol table only those global symbols that are defined in the specified file and are currently unresolved. If /SELECTIVE_SEARCH is not specified, the linker includes all symbols from that file in its global symbol table.

/SHAREABLE

Requests that the linker include a shareable image file. No wildcard characters are allowed in the file specification.

The following table contains the abbreviations of the qualifiers for the .LINK directive. Note that to ensure readability, as well as compatibility with future releases, it is recommended that you use the full names of the qualifiers.

Abbreviation	Qualifier
/I	/INCLUDE
/L	/LIBRARY
/SE	/SELECTIVE_SEARCH
/SH	/SHAREABLE

DESCRIPTION

The .LINK directive allows you to include linker option records in an object module produced by VAX MACRO. The qualifiers for the .LINK directive perform functions similar to the functions performed by the same qualifiers for the DCL command LINK.

You should use the .LINK directive for references that are not linker defaults, but that you always want to include in a particular image. Using the .LINK directive enables you to avoid having to explicitly name these references in the DCL command LINK.

For detailed information on the qualifiers to the DCL command LINK, see the *VMS DCL Dictionary*. For a complete discussion of the operation of the linker itself, see the *VMS Linker Utility Manual.*

EXAMPLES

1
```
.LINK "SYS$LIBRARY:MYLIB" /INCLUDE=(MOD1, MOD2, MOD6)
```

This statement, when included in the file MYPROG.MAR, causes the assembler to request that MYPROG.OBJ be linked with modules MOD1, MOD2, and MOD6 in the library SYS$LIBRARY:MYLIB.OLB (where SYS$LIBRARY is a logical name for the disk and directory in which MYLIB.OLB is listed). The library is not searched for other unresolved references. The statement is equivalent to linking the file with the DCL command:

```
2    $ LINK MYPROG, SYS$LIBRARY:MYLIB /INCLUDE=(MOD1, MOD2, MOD6)


3    .LINK \SYS$LIBRARY:MYOBJ\                          ; Link with object module
                                                        ;   SYS$LIBRARY:MYOBJ.OBJ

     .LINK 'SYS$LIBRARY:YOURLIB' /LIBRARY               ; Search object library
                                                        ;   SYS$LIBRARY:YOURLIB.OLB
                                                        ;   for unresolved references

     .LINK *SYS$LIBRARY:MYSTB.STB* /SELECTIVE_SEARCH ; Search symbol table
                                                        ;   SYS$LIBRARY:MYSTB.STB
                                                        ;   for unresolved references

     .LINK "SYS$LIBRARY:MYSHR.EXE" /SHAREABLE           ; Link with shareable image
                                                        ;   SYS$LIBRARY:MYSHR.EXE
```

To increase efficiency and performance, include several related input files in a single .LINK directive. The following example shows how the five options illustrated previously can be included in one statement:

```
4    .LINK   'SYS$LIBRARY:MYOBJ',-
             'SYS$LIBRARY:YOURLIB' /LIBRARY,-
             'SYS$LIBRARY:MYLIB' /INCLUDE=(MOD1, MOD2, MOD6),-
             'SYS$LIBRARY:MYSTB.STB' /SELECTIVE_SEARCH,-
             'SYS$LIBRARY:MYSHR.EXE' /SHAREABLE
```

.LIST

Listing directive

FORMAT **.LIST** *[argument-list]*

PARAMETER ***argument-list***

One or more of the symbolic arguments defined in Table 6–8. You can use either the long form or the short form of the arguments. If multiple arguments are specified, separate them with commas (,), spaces, or tabs.

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

.LONG

Longword storage directive

FORMAT

.LONG *expression-list*

PARAMETERS

expression-list
One or more expressions separated by commas (,). You have the option of following each expression with a repetition factor delimited by square brackets ([]).

An expression followed by a repetition factor has the format:

expression1[expression2]

expression1
An expression that specifies the value to be stored.

[expression2]
An expression that specifies the number of times the value is repeated. The expression must not contain any undefined symbols and must be an absolute expression (see Section 3.5). The square brackets are required.

DESCRIPTION

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

EXAMPLE

```
LAB_3:  .LONG   LAB_3,^X7FFFFFFF,^A'ABCD' ; 3 longwords of data
        .LONG   ^XF@4                     ; 1 longword of data
        .LONG   0[22]                     ; 22 longwords of data
```

Note

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

.MACRO

Macro definition directive

FORMAT

```
.MACRO   macro-name [formal-argument-list]
   .
   .
   .
range
   .
   .
   .
.ENDM   [macro name]
```

PARAMETERS

macro-name
The name of the macro to be defined; this name can be any legal symbol up to 31 characters long.

formal-argument-list
The symbols, separated by commas (,), to be replaced by the actual arguments in the macro call.

range
The source text to be included in the macro expansion.

DESCRIPTION

.MACRO begins the definition of a macro. It gives the macro name and a list of formal arguments (see Chapter 4). If the name specified is the same as the name of a previously defined macro, the previous definition is deleted and replaced with the new one. The .MACRO directive is followed by the source text to be included in the macro expansion. The .ENDM directive specifies the end of the range.

Macro names do not conflict with user-defined symbols. Both a macro and a user-defined symbol can have the same name.

When the assembler encounters a .MACRO directive, it adds the macro name to its macro name table and stores the source text of the macro (up to the matching .ENDM directive). No other processing occurs until the macro is expanded.

The symbols in the formal argument list are associated with the macro name and are limited to the scope of the definition of that macro. For this reason, the symbols that appear in the formal argument list can also appear elsewhere in the program.

Notes

1 If a macro has the same name as a VAX opcode, the macro is used instead of the instruction. This feature allows you to temporarily redefine an opcode.

2 If a macro has the same name as a VAX opcode and is in a macro library, you must use the .MCALL directive to define the macro. Otherwise, because the symbol is already defined (as the opcode), the assembler will not search the macro libraries.

3 You can redefine a macro with new source text during assembly by specifying a second .MACRO directive with the same name. Including a second .MACRO directive within the original macro definition causes the first macro call to redefine the macro. This feature is useful when a macro performs initialization or defines symbols, when an operation is performed only once. The macro redefinition can eliminate unneeded source text in a macro or it can delete the entire macro. The .MDELETE directive provides another way to delete macros.

EXAMPLE

The macro definition is as follows:

```
        .MACRO  USERDEF
        .PSECT  DEFIES,ABS
MYSYM=  5
HIVAL=  ^XFFF123
LOWVAL= 0
        .PSECT  RWDATA,NOEXE,LONG
TABLE:  .BLKL   100
LIST:   .BLKB   10
        .MACRO  USERDEF                 ; Redefine it to null
        .ENDM   USERDEF
        .ENDM   USERDEF
```

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

```
        USERDEF                         ; Should expand data
        .PSECT  DEFIES,ABS
MYSYM=  5
HIVAL=  ^XFFF123
LOWVAL= 0
        .PSECT  RWDATA,NOEXE,LONG
TABLE:  .BLKL   100
LIST:   .BLKB   10
        .MACRO  USERDEF                 ; Redefine it to null
        .ENDM   USERDEF

        USERDEF                         ; Should expand nothing
```

In this example, when the macro is called the first time, it defines some symbols and data storage areas and then redefines itself. When the macro is called a second time, the macro expansion contains no source text.

.MASK

Mask directive

FORMAT **.MASK** *symbol[,expression]*

PARAMETERS

symbol
A symbol defined in an .ENTRY directive.

expression
A register save mask.

DESCRIPTION

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

Notes

1. If .MASK does not contain an expression, the assembler directs the linker to copy the register save mask specified in .ENTRY to the word reserved by .MASK.
2. If .MASK contains an expression, the assembler directs the linker to combine this expression with the register save mask specified in .ENTRY and store the result in the word reserved by .MASK. The linker performs an inclusive OR operation to combine the mask in the entry point and the value of the expression. Consequently, a register specified in either .ENTRY or .MASK will be included in the combined mask. See the description of .ENTRY for more information on entry masks.

.MCALL

Macro call directive

FORMAT

.MCALL *macro-name-list*

PARAMETERS

macro-name-list
A list of macros to be defined for this assembly. Separate the macro names with commas (,).

DESCRIPTION

.MCALL specifies the names of the system and user-defined macros that are required to assemble the source program but are not defined in the source file.

If any named macro is not found upon completion of the search (that is, if the macro is not defined in any of the macro libraries), the assembler displays an error message.

Note: **.MCALL is provided for compatibility with MACRO–11; with one exception, Digital recommends that you not use it. When VAX MACRO finds an unknown symbol in the opcode field, it automatically searches all macro libraries. If it finds the symbol in a library, it uses the macro definition and expands the macro reference. If VAX MACRO does not find the symbol in the library, it displays an error message.**

You must use .MCALL when a macro has the same name as an opcode (see description of .MACRO).

EXAMPLE

```
.MCALL   INSQUE          ; Substitute macro in
                         ;   library for INSQUE
                         ;   instruction
```

.MDELETE

Macro deletion directive

FORMAT

.MDELETE *macro-name-list*

PARAMETERS

macro-name-list
A list of macros whose definitions are to be deleted. Separate the names with commas (,).

DESCRIPTION

.MDELETE deletes the definitions of specified macros. The number of macros actually deleted is printed in the assembly listing on the same line as the .MDELETE directive.

.MDELETE completely deletes the macro, freeing memory as necessary. Macro redefinition with .MACRO merely redefines the macro.

EXAMPLE

```
.MDELETE      USERDEF,$SSDEF,ALTR
```

.MEXIT

Macro exit directive

FORMAT

.MEXIT

DESCRIPTION

.MEXIT terminates a macro expansion before the end of the macro. Termination is the same as if .ENDM were encountered. You can use the directive within repeat blocks. .MEXIT is useful in conditional expansion of macros because it bypasses the complexities of nested conditional directives and alternate assembly paths.

Notes

1. When .MEXIT occurs in a repeat block, the assembler terminates the current repetition of the range and suppresses further expansion of the repeat range.
2. When macros or repeat blocks are nested, .MEXIT exits to the next higher level of expansion.
3. If .MEXIT occurs outside a macro definition or a repeat block, the assembler displays an error message.

EXAMPLE

```
.MACRO   POLO    N,A,B
   .
   .
   .
.IF EQ  N                   ; Start conditional assembly block
   .
   .
   .
.MEXIT                      ; Terminate macro expansion
.ENDC                       ; End conditional assembly block
   .
   .
   .
.ENDM   POLO                ; Normal end of macro
```

In this example, if the actual argument for the formal argument N equals zero, the conditional block is assembled, and the macro expansion is terminated by .MEXIT.

.NARG

Number of arguments directive

FORMAT

.NARG *symbol*

PARAMETERS

symbol
A symbol that is assigned a value equal to the number of arguments in the macro call.

DESCRIPTION

.NARG determines the number of arguments in the current macro call.

.NARG counts all the positional arguments specified in the macro call, including null arguments (specified by adjacent commas (,)). The value assigned to the specified symbol does not include either any keyword arguments or any formal arguments that have default values.

Note

If .NARG appears outside a macro, the assembler displays an error message.

EXAMPLE

The macro definition is as follows:

```
.MACRO  CNT_ARG A1,A2,A3,A4,A5,A6,A7,A8,A9=DEF9,A10=DEF10
.NARG   COUNTER          ; COUNTER is set to no. of ARGS
.WORD   COUNTER          ; Store value of COUNTER
.ENDM   CNT_ARG
```

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

```
CNT_ARG TEST,FIND,ANS    ; COUNTER will = 3
.NARG   COUNTER          ; COUNTER is set to no. of ARGS
.WORD   COUNTER          ; Store value of COUNTER

CNT_ARG                  ; COUNTER will = 0
.NARG   COUNTER          ; COUNTER is set to no. of ARGS
.WORD   COUNTER          ; Store value of COUNTER

CNT_ARG TEST,A2=SYMB2,A3=SY3      ; COUNTER will = 1
.NARG   COUNTER          ; COUNTER is set to no. of ARGS
.WORD   COUNTER          ; Store value of COUNTER
                         ; Keyword arguments are not counted

CNT_ARG ,SYMBL,,         ; COUNTER will = 3
.NARG   COUNTER          ; COUNTER is set to no. of ARGS
.WORD   COUNTER          ; Store value of COUNTER
                         ; Null arguments are counted
```

.NCHR

Number of characters directive

FORMAT

.NCHR *symbol,<string>*

PARAMETERS

symbol

A symbol that is assigned a value equal to the number of characters in the specified character string.

<string>

A sequence of printable characters. Delimit the character string with angle brackets (<>) (or a character preceded by a circumflex (^)) only if the specified character string contains a legal separator (comma (,), space, and/or tab) or a semicolon (;).

DESCRIPTION

.NCHR determines the number of characters in a specified character string. It can appear anywhere in a VAX MACRO program and is useful in calculating the length of macro arguments.

EXAMPLE

The macro definition is as follows:

```
.MACRO   CHAR    MESS                  ; Define MACRO
.NCHR    CHRCNT,<MESS>                 ; Assign value to CHRCNT
.WORD    CHRCNT                        ; Store value
.ASCII   /MESS/                        ; Store characters
.ENDM    CHAR                          ; Finish
```

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

```
CHAR     <HELLO>                      ; CHRCNT will = 5
.NCHR    CHRCNT,<HELLO>               ; Assign value to CHRCNT
.WORD    CHRCNT                        ; Store value
.ASCII   /HELLO/                       ; Store characters

CHAR     <14, 75.39  4>               ; CHRCNT will = 12(dec)
.NCHR    CHRCNT,<14, 75.39  4>        ; Assign value to CHRCNT
.WORD    CHRCNT                        ; Store value
.ASCII   /14, 75.39  4/                ; Store characters
```

.NLIST

Listing directive

FORMAT **.NLIST** *[argument-list]*

PARAMETER ***argument-list***
One or more of the symbolic arguments listed in Table 6–8. Use either the long form or the short form of the arguments. If you specify multiple arguments, separate them with commas (,), spaces, or tabs.

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

.NOCROSS

Cross-reference directive

FORMAT

.NOCROSS *[symbol-list]*

PARAMETER

symbol-list
A list of legal symbol names separated by commas (,).

DESCRIPTION

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

.NOSHOW

Listing directive

FORMAT **.NOSHOW** *[argument-list]*

PARAMETER

argument-list
One or more of the symbolic arguments listed in Table 6–8 in the description of .SHOW. Use either the long form or the short form of the arguments. If you specify multiple arguments, separate them with commas (,), spaces, or tabs.

DESCRIPTION

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

.NTYPE

Operand type directive

FORMAT **.NTYPE** *symbol,operand*

PARAMETERS

symbol
Any legal VAX MACRO symbol. This symbol is assigned a value equal to the 8- or 16-bit addressing mode of the operand argument that follows.

operand
Any legal address expression, as you use it with an opcode. If no argument is specified, zero is assumed.

DESCRIPTION

.NTYPE determines the addressing mode of the specified operand.

The value of the symbol is set to the specified addressing mode. In most cases, an 8-bit (1-byte) value is returned. Bits 0 to 3 specify the register associated with the mode, and bits 4 to 7 specify the addressing mode. To provide concise addressing information, the mode bits 4 to 7 are not exactly the same as the numeric value of the addressing mode described in Table C–6. Literal mode is indicated by a zero in bits 4 to 7, instead of the values 0 to 3. Mode 1 indicates an immediate mode operand, mode 2 indicates an absolute mode operand, and mode 3 indicates a general mode operand.

For indexed addressing mode, a 16-bit (2-byte) value is returned. The high-order byte contains the addressing mode of the base operand specifier and the low-order byte contains the addressing mode of the primary operand (the index register).

See Chapter 5 of this volume for more information on addressing modes.

EXAMPLE

```
; The following macro is used to push an address on the stack.  It checks
; the operand type (by using .NTYPE) to determine if the operand is an
; address and, if not, the macro simply pushes the argument on the stack
; and generates a warning message.
;
        .MACRO  PUSHADR #ADDR
        .NTYPE  A,ADDR                   ; Assign operand type to 'A'
A = A@-4&^XF                             ; Isolate addressing mode
        .IF IDENTICAL 0,<ADDR>           ; Is argument exactly 0?
        PUSHL   #0                       ; Stack zero
        .MEXIT                           ; Exit from macro
        .ENDC
ERR = 0                                  ; ERR tells if mode is address
                                         ; ERR = 0 if address, 1 if not
        .IIF LESS_EQUAL A-1,    ERR=1    ; Is mode not literal or immediate?
        .IIF EQUAL A-5,    ERR=1         ; Is mode not register?
        .IF EQUAL  ERR                   ; Is mode address?
        PUSHAL  ADDR                     ; Yes, stack address
        .IFF                             ; No
        PUSHL   ADDR                     ; Then stack operand & warn
        .WARN   ; ADDR is not an address;
        .ENDC
        .ENDM   PUSHADR
```

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

```
        PUSHADR (R0)                     ; Valid argument
        PUSHAL  (R0)                     ; Yes, stack address

        PUSHADR (R1)[R4]                 ; Valid argument
        PUSHAL  (R1)[R4]                 ; Yes, stack address

        PUSHADR 0                        ; Is zero
        PUSHL   #0                       ; Stack zero

        PUSHADR #1                       ; Not an address
        PUSHL   #1                       ; Then stack operand & warn
%MACRO-W-GENWRN, Generated WARNING: #1 is not an address

        PUSHADR R0                       ; Not an address
        PUSHL   R0                       ; Then stack operand & warn
%MACRO-W-GENWRN, Generated WARNING: R0 is not an address
```

Note that to save space, this example is listed as it would appear if .SHOW BINARY, not .SHOW EXPANSIONS, were specified in the source program.

.OCTA

Octaword storage directive

FORMAT

.OCTA *literal*
.OCTA *symbol*

PARAMETERS

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

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

DESCRIPTION

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

Note

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

EXAMPLE

```
.OCTA   ^A"FEDCBA987654321"         ; Each ASCII character
                                    ;   is stored in a byte
.OCTA   0                           ; OCTA 0
.OCTA   ^X01234ABCD5678F9           ; OCTA hex value specified
.OCTA   VINTERVAL                   ; VINTERVAL has 32-bit value,
                                    ;   sign-extended
```

.ODD

Odd location counter alignment directive

FORMAT **.ODD**

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

.OPDEF

Opcode definition directive

FORMAT

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

PARAMETERS

opcode

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

value

An expression that specifies the value of the opcode. The expression must be absolute and must not contain any undefined values (see Section 3.5). The value of the expression must be in the range 0 to $65,535_{10}$ (hexadecimal FFFF), but you cannot use the values 252 to 255 because the architecture specifies these as the start of a 2-byte opcode. The expression is represented as follows:

If 0 < expression < 251	Expression is a 1-byte opcode.
If expression > 255	Expression bits 7:0 are the first byte of the opcode and expression bits 15:8 are the second byte of the opcode.

operand-descriptor-list

A list of operand descriptors that specifies the number of operands and the type of each. Up to 16 operand descriptors are allowed in the list. Table 6–5 lists the operand descriptors.

Table 6–5 Operand Descriptors

Access Type	Data Type								
	Byte	Word	Long-word	Floating Point	Double Floating Point	G_ Floating Point	H_ Floating Point	Quad-word	Octa-word
Address	AB	AW	AL	AF	AD	AG	AH	AQ	AO
Read-only	RB	RW	RL	RF	RD	RG	RH	RQ	RO
Modify	MB	MW	ML	MF	MD	MG	MH	MQ	MO
Write-only	WB	WW	WL	WF	WD	WG	WH	WQ	WO

(continued on next page)

Table 6–5 (Cont.) Operand Descriptors

Access Type	Data Type								
	Byte	Word	Long-word	Float-ing Point	Double Float-ing Point	G_ Floating Point	H_ Floating Point	Quad-word	Octa-word
Field	VB	VW	VL	VF	VD	VG	VH	VQ	VO
Branch	BB	BW	—	—	—	—	—	—	—

DESCRIPTION

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

Notes

1 You can also use a macro to redefine an opcode (see the description of .MACRO in this section). Note that the macro name table is searched before the user-defined opcode table.

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

3 The operand descriptors are specified in a format similar to the operand specifier notation described in Chapter 8. The first character specifies the operand access type, and the second character specifies the operand data type.

EXAMPLE

```
.OPDEF  MOVL3   ^XA9FF,RL,ML,WL          ; Defines an instruction
                                         ;   MOVL3, which uses
                                         ;   the reserved opcode FF
.OPDEF  DIVF2   ^X46,RF,MF               ; Redefines the DIVF2 and
.OPDEF  MOVC5   ^X2C,RW,AB,AB,RW,AB      ;   MOVC5 instructions

.OPDEF  CALL    ^X10,BB                  ; Equivalent to a BSBB
```

.PACKED

Packed decimal string storage directive

FORMAT **.PACKED** *decimal-string[,symbol]*

PARAMETERS

decimal-string
A decimal number from 0 to 31 digits long with an optional sign. Digits can be in the range 0 to 9 (see Section 8.2.14).

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

DESCRIPTION .PACKED generates packed decimal data, two digits per byte. Packed decimal data is useful in calculations requiring exact accuracy. Packed decimal data is operated on by the decimal string instructions. See Section 8.2.14 for more information on the format of packed decimal data.

EXAMPLE

```
.PACKED -12,PACK_SIZE          ; PACK_SIZE gets value of 2
.PACKED +500
.PACKED 0
.PACKED -0,SUM_SIZE            ; SUM_SIZE gets value of 1
```

.PAGE

Page ejection directive

FORMAT

.PAGE

DESCRIPTION

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

VAX MACRO ignores .PAGE in a macro definition. The paging operation is performed only during macro expansion.

.PRINT

Assembly message directive

FORMAT

.PRINT *[expression] ;comment*

PARAMETERS

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

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

DESCRIPTION

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

Notes

1. .PRINT, .ERROR, and .WARN are called the message display directives. You can use these to display information indicating that a macro call contains an error or an illegal set of conditions.
2. If .PRINT is included in a macro library, end the comment with an additional semicolon. If you omit the semicolon, the comment will be stripped from the directive and will not be displayed when the macro is called.
3. If the expression has a value of zero, it is not displayed with the message.

EXAMPLE

```
.PRINT  2       ; The sine routine has been changed
```

.PSECT

Program sectioning directive

FORMAT **.PSECT** *[program-section-name[,argument-list]]*

PARAMETERS

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

argument-list
A list containing the program section attributes and the program section alignment. Table 6–6 lists the attributes and their functions. Table 6–7 lists the default attributes and their opposites. Program sections are aligned when you specify an integer in the range 0 to 9 or one of the five keywords listed in the following table. If you specify an integer, the program section is linked to begin at the next virtual address, which is a multiple of 2 raised to the power of the integer. If you specify a keyword, the program section is linked to begin at the next virtual address (a multiple of the values listed in the following table):

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

BYTE is the default.

Table 6–6 Program Section Attributes

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

(continued on next page)

Table 6–6 (Cont.) Program Section Attributes

Attribute	Function
PIC	Position-Independent Content—The program section can be relocated; that is, it can be assigned to any memory area (when it is in a shareable image).
RD	Readable—Reserved for future use.
REL	Relocatable—The linker assigns the program section a relocatable base address. The contents of the program section can be code or data. Compare this attribute with its opposite, ABS.
SHR	Share—The program section can be shared at execution time by multiple processes. This attribute is assigned to a program section that can be linked into a shareable image.
USR	User Segment—Reserved for future use.
VEC	Vector-Containing—The program section contains a change mode vector indicating a privileged shareable image. You must use the SHR attribute with VEC.
WRT	Write—The contents of the program section can be altered (written into) at execution time.

Table 6–7 Default Program Section Attributes

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

DESCRIPTION

.PSECT defines a program section and its attributes and refers to a program section once it is defined. Use program sections to do the following:

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

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

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

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

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

The attributes listed in the .PSECT directive only describe the contents of the program section. The assembler does not check to ensure that the contents of the program section actually include the attributes listed. However, the assembler and the linker do check that all program sections with the same name have exactly the same attributes. The assembler and linker display an error message if the program section attributes are not consistent.

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

Notes

1. The .ALIGN directive cannot specify an alignment greater than that of the current program section; consequently, .PSECT should specify the largest alignment needed in the program section. For efficiency of execution, an alignment of longword or larger is recommended for all program sections that have longword data.
2. The attributes of the default absolute and the default unnamed program sections are listed in the following table. Note that the program section names include the periods (.) and enclosed spaces.

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

EXAMPLE

```
.PSECT   CODE,NOWRT,EXE,LONG       ; Program section to contain
                                   ;   executable code
.PSECT   RWDATA,WRT,NOEXE,QUAD
                                   ; Program section to contain
                                   ;   modifiable data
```

.QUAD

Quadword storage directive

FORMAT

.QUAD *literal*
.QUAD *symbol*

PARAMETERS

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

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

DESCRIPTION

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

Note

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

EXAMPLE

```
.QUAD   ^A'..ASK?..'          ; Each ASCII character is stored
                              ;   in a byte
.QUAD   0                     ; QUAD 0
.QUAD   ^X0123456789ABCDEF    ; QUAD hex value specified
.QUAD   ^B1111000111001101    ; QUAD binary value specified
.QUAD   LABEL                 ; LABEL has a 32-bit,
                              ;   zero-extended value.
```

.REFn

Operand generation directives

FORMAT

.REF1 *operand*
.REF2 *operand*
.REF4 *operand*
.REF8 *operand*
.REF16 *operand*

PARAMETER

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

DESCRIPTION

VAX MACRO has the following five operand generation directives that you can use in macros to define new opcodes:

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

The .REFn directives are provided for compatibility with VAX MACRO Version 1.0. Because the .OPDEF directive provides greater functionality and is easier to use than .REFn, you should use .OPDEF instead of .REFn.

EXAMPLE

```
.MACRO  MOVL3 A,B,C
.BYTE   ^XFF,^XA9
.REF4   A                  ; This operand has longword context
.REF4   B                  ; This operand has longword context
.REF4   C                  ; This operand has longword context
.ENDM   MOVL3

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

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

.REPEAT

Repeat block directive

FORMAT

```
.REPEAT expression
   .
   .
   .
range
   .
   .
   .
.ENDR
```

PARAMETERS

expression
An expression whose value controls the number of times the range is to be assembled within the program. When the expression is less than or equal to zero, the repeat block is not assembled. The expression must be absolute and must not contain any undefined symbols (see Section 3.5).

range
The source text to be repeated the number of times specified by the value of the expression. The repeat block can contain macro definitions, indefinite repeat blocks, or other repeat blocks. .MEXIT is legal within the range.

DESCRIPTION

.REPEAT repeats a block of code a specified number of times, in line with other source code. The .ENDR directive specifies the end of the range.

Note

The alternate form of .REPEAT is .REPT.

EXAMPLE

The macro definition is as follows:

```
.MACRO  COPIES  STRING,NUM
.REPEAT NUM
.ASCII  /STRING/
.ENDR
.BYTE   0
.ENDM   COPIES
```

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

```
        COPIES  <ABCDEF>,5
        .REPEAT 5
        .ASCII  /ABCDEF/
        .ENDR
        .ASCII  /ABCDEF/
        .ASCII  /ABCDEF/
        .ASCII  /ABCDEF/
        .ASCII  /ABCDEF/
        .ASCII  /ABCDEF/
        .BYTE   0

VARB = 3
        COPIES  <HOW MANY TIMES>,VARB
        .REPEAT 3
        .ASCII  /HOW MANY TIMES/
        .ENDR
        .ASCII  /HOW MANY TIMES/
        .ASCII  /HOW MANY TIMES/
        .ASCII  /HOW MANY TIMES/
        .BYTE   0
```

.RESTORE_PSECT

Restore previous program section context directive

FORMAT

.RESTORE_PSECT

DESCRIPTION

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

Note

The alternate form of .RESTORE_PSECT is .RESTORE.

EXAMPLE

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

```
        .MACRO  INITD                          ; Initialize symbols
                                               ;   and data areas
        .SAVE_PSECT                            ; Save the current PSECT
        .PSECT  SYMBOLS,ABS                    ; Define new PSECT
HELP_LEV=2                                     ; Define symbol
MAXNUM=100                                     ; Define symbol
RATE1=16                                       ; Define symbol
RATE2=4                                        ; Define symbol
        .PSECT  DATA,NOEXE,LONG                ; Define another PSECT
TABL:   .BLKL   100                            ; 100 longwords in TABL
TEMP:   .BLKB   16                             ; More storage
        .RESTORE_PSECT                         ; Restore the PSECT
                                               ;   in effect when
                                               ;   MACRO is invoked
        .ENDM
```

.SAVE_PSECT

Save current program section context directive

FORMAT

.SAVE_PSECT [LOCAL_BLOCK]

PARAMETER

LOCAL_BLOCK

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

DESCRIPTION

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

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

Note

The alternate form of .SAVE_PSECT is .SAVE.

EXAMPLE

```
The macro definition is as follows:
        .MACRO  ERR_MESSAGE,TEXT            ; Set up lists of messages
                                            ;   and pointers
                                            ;
        .IIF    NOT_DEFINED      MESSAGE_INDEX, MESSAGE_INDEX=0
        .SAVE_PSECT -
                LOCAL_BLOCK                 ; Keep local labels
        .PSECT  MESSAGE_TEXT                ; List of error messages
MESSAGE::
        .ASCIC  /TEXT/
        .PSECT  MESSAGE_POINTERS            ; Addresses of error
        .ADDRESS -                          ;   messages
                MESSAGE                     ; Store one pointer
        .RESTORE_PSECT                      ; Get back local labels
        PUSHL   #MESSAGE_INDEX              ;
        CALLS   #1,PRINT_MESS               ; Print message
MESSAGE_INDEX=MESSAGE_INDEX+1
        .ENDM   ERR_MESSAGE
```

```
Macro call:

RESETS: CLRL    R4
        BLBC    R0,30$
        ERR_MESSAGE     <STRING TOO SHORT> ; Add "STRING TOO SHORT"
                                           ;   to list of error
30$:    RSB                                ;   messages
```

By using .SAVE_PSECT LOCAL_BLOCK, the local label 30$ is defined in the same local label block as the reference to 30$. If a local label is not defined in the block in which it is referenced, the assembler produces the following error message:

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

.SHOW
.NOSHOW

Listing directives

FORMAT

.SHOW *[argument-list]*
.NOSHOW *[argument-list]*

PARAMETER

argument-list
One or more of the optional symbolic arguments defined in Table 6–8. You can use either the long form or the short form of the arguments. You can use each argument alone or in combination with other arguments. If you specify multiple arguments, you must separate them by commas (,), tabs, or spaces. If any argument is not specifically included in a listing control statement, the assembler assumes its default value (SHOW or NOSHOW) throughout the source program.

Table 6–8 .SHOW and .NOSHOW Symbolic Arguments

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

DESCRIPTION

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

When you use them with an argument list, .SHOW includes and .NOSHOW excludes the lines specified in Table 6–8. .SHOW and .NOSHOW control the listing of the source lines that are in conditional assembly blocks (see the description of .IF), macros, and repeat blocks.

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

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

Notes

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

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

EXAMPLE

```
        .MACRO  XX
          .
          .
          .
        .SHOW                   ; List next line
X=.
        .NOSHOW                 ; Do not list remainder
          .                     ;   of macro expansion
          .
          .
        .ENDM
        .NOSHOW EXPANSIONS      ; Do not list macro
                                ;   expansions
        XX
X=.
```

.SIGNED_BYTE

Signed byte data directive

FORMAT

.SIGNED_BYTE *expression-list*

PARAMETERS

expression-list

An expression or list of expressions separated by commas (,). You have the option of following each expression with a repetition factor delimited by square brackets ([ ]).

An expression followed by a repetition factor has the format:

expression1 [expression2]

expression1

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

[expression2]

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

DESCRIPTION

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

Note

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

EXAMPLE

```
.SIGNED_BYTE   LABEL1-LABEL2   ;  Data must fit
.SIGNED_BYTE   ALPHA[20]       ;    in byte
```

.SIGNED_WORD

Signed word storage directive

FORMAT

.SIGNED_WORD *expression-list*

PARAMETERS

expression-list

An expression or list of expressions separated by commas (,). You have the option of following each expression with a repetition factor delimited by square brackets ([ ]).

An expression followed by a repetition factor has the format:

expression1 [expression2]

expression1

An expression that specifies the value to be stored. The value must be in the range –32,768 to +32,767.

[expression2]

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

DESCRIPTION

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

Note

Specifying .SIGNED_WORD allows the linker to detect overflow conditions when the value of the expression is in the range of 32,768 to 65,535. Values in this range can be stored as unsigned data but cannot be stored as signed data in a word.

EXAMPLE

```
        .MACRO   CASE,SRC,DISPLIST,TYPE=W,LIMIT=#0,NMODE=S^#,?BASE,?MAX
                                        ; MACRO to use CASE instruction,
                                        ;   SRC is selector, DISPLIST
                                        ;   is list of displacements, TYPE
                                        ;   is B (byte) W (word) L (long),
                                        ;   LIMIT is base value of selector
        CASE'TYPE       SRC,LIMIT,NMODE'<<MAX-BASE>/2>-1
                                        ; Case instruction
BASE:                                   ; Local label specifying base
        .IRP    EP,<DISPLIST>           ;   to set up offset list
        .SIGNED_WORD   EP-BASE          ; Offset list
        .ENDR                           ;
MAX:                                    ; Local label used to count
        .ENDM   CASE                    ;   args

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

30000$:                                 ; Local label specifying base
        .SIGNED_WORD    ERR_PROC-30000$ ; Offset list
        .SIGNED_WORD    SORT-30000$     ; Offset list
        .SIGNED_WORD    REV_SORT-30000$ ; Offset list
30001$:                                 ; Local label used to count args
                                        ; =1, forward sort;  =2, backward
                                        ;   sort

        CASE    TEST    <TEST1,TEST2,TEST3>,L,#1
        CASEL   TEST,#1,S^#<<30003$-30002$>/2>-1
30002$:                                 ; Local label specifying base
        .SIGNED_WORD    TEST1-30002$    ; Offset list
        .SIGNED_WORD    TEST2-30002$    ; Offset list
        .SIGNED_WORD    TEST3-30002$    ; Offset list
30003$:                                 ; Local label used to count args
                                        ; Value of TEST can be 1, 2, or 3
```

In this example, the CASE macro uses .SIGNED_WORD to create a CASEB, CASEW, or CASEL instruction.

.SUBTITLE

Subtitle directive

FORMAT

.SUBTITLE *comment-string*

PARAMETER

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

DESCRIPTION

.SUBTITLE causes the assembler to print the line of text, represented by the comment-string, in the table of contents (which the assembler produces immediately before the assembly listing). The assembler also prints the line of text as the subtitle on the second line of each assembly listing page. This subtitle text is printed on each page until altered by a subsequent .SUBTITLE directive in the program.

Note

The alternate form of .SUBTITLE is .SBTTL.

EXAMPLES

1
```
.SUBTITLE CONDITIONAL ASSEMBLY
```

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

```
CONDITIONAL ASSEMBLY
```

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

2
```
TABLE OF CONTENTS
```

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

.TITLE

Title directive

FORMAT

.TITLE *module-name comment-string*

PARAMETERS

module-name
An identifier from 1 to 31 characters long.

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

DESCRIPTION

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

Notes

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

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

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

EXAMPLE

```
.TITLE  EVAL    Evaluates Expressions
```

.TRANSFER

Transfer directive

FORMAT **.TRANSFER** *symbol*

PARAMETER

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

DESCRIPTION

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

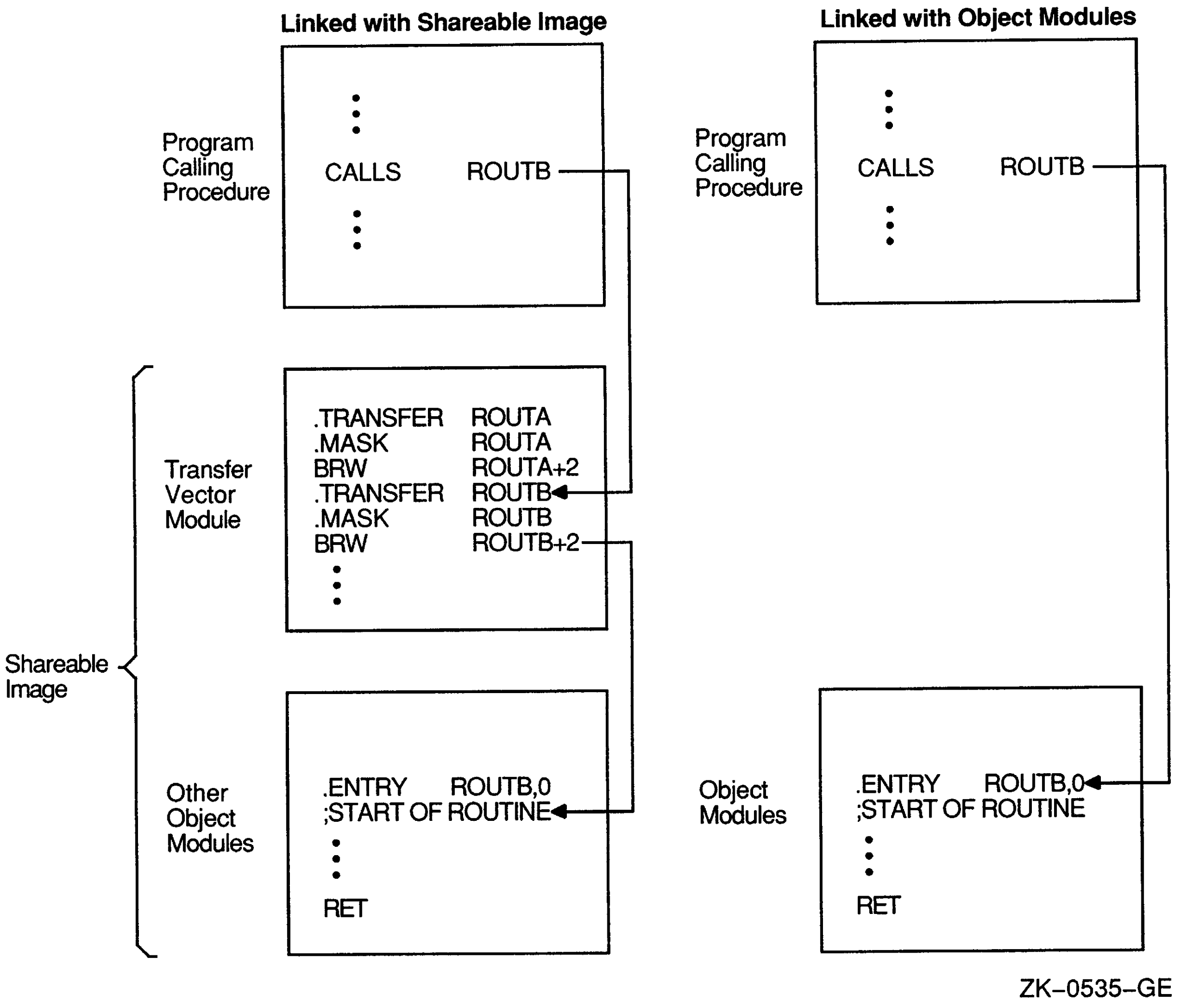
To make program maintenance easier, programs should not need to be relinked when the shareable images to which they are linked change. To avoid relinking entire programs when their linked shareable images change, keep the entry points in the changed shareable image at their original addresses. To do this, create an object module that contains a transfer vector for each entry point. Do not change the order of the transfer vectors. Link this object module at the beginning of the shareable image. The addresses of the entry points remain fixed even if the source code for a routine is changed. After each .TRANSFER directive, create a register save mask (for procedures only) and a branch to the first instruction of the routine.

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

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

Figure 6–1 illustrates the use of transfer vectors.

Figure 6–1 Using Transfer Vectors



EXAMPLE

```
.TRANSFER ROUTINE_A
.MASK     ROUTINE_A,^M<R4,R5>   ; Copy entry mask
                                ;   and add registers
                                ;   R4 and R5
BRW       ROUTINE_A+2           ; Branch to routine
                                ;   (past entry mask)
   .
   .
   .
.ENTRY    ROUTINE_A,^M<R2,R3>   ; ENTRY point, save
                                ;   registers R2 and R3
   .
   .
   .
RET
```

In this example, .MASK copies the entry mask of a routine to the new entry address specified by .TRANSFER. If the routine is placed in a shareable image and then called, registers R2, R3, R4, and R5 will be saved.

.WARN

Warning directive

FORMAT

.WARN *[expression] ;comment*

PARAMETERS

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

;comment
A comment that is displayed when .WARN is encountered during assembly. The comment must be preceded by a semicolon (;).

DESCRIPTION

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

Notes

1. .WARN, .ERROR, and .PRINT are called the message display directives. Use them to display information indicating that a macro call contains an error or an illegal set of conditions.
2. When the assembly is finished, the assembler displays on the terminal or in the batch log file, the total number of errors, warnings, and information messages, and the page numbers and line numbers of the lines causing the errors or warnings.
3. If .WARN is included in a macro library, end the comment with an additional semicolon. If you omit the semicolon, the comment will be stripped from the directive and will not be displayed when the macro is called.
4. The line containing the .WARN directive is not included in the listing file.
5. If the expression has a value of zero, it is not displayed in the warning message.

EXAMPLE

```
 .IF DEFINED   FULL
 .IF DEFINED   DOUBLE_PREC
 .WARN         ; This combination not tested
 .ENDC
 .ENDC
```

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

```
%MACRO-W-GENWRN, Generated WARNING: This combination not tested
```

.WEAK

Weak symbol attribute directive

FORMAT

.WEAK *symbol-list*

PARAMETER

symbol-list
A list of legal symbols separated by commas (,).

DESCRIPTION

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

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

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

EXAMPLE

```
.WEAK     IOCAR,LAB_3
```

.WORD

Word storage directive

FORMAT

.WORD *expression-list*

PARAMETERS

expression-list

One or more expressions separated by commas (,). You have the option of following each expression by a repetition factor delimited with square brackets ([ ]).

An expression followed by a repetition factor has the format:

expression1[expression2]

expression1

An expression that specifies the value to be stored.

[expression2]

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

DESCRIPTION

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

Notes

1. The expression is first evaluated as a longword, then truncated to a word. The value of the expression should be in the range of –32,768 to +32,767 for signed data or 0 to 65,535 for unsigned data. The assembler displays an error if the high-order 2 bytes of the longword expression have a value other than zero or ^XFFFF.
2. The .SIGNED_WORD directive is the same as .WORD except that the assembler displays a diagnostic message if a value is in the range from 32,768 to 65,535.

EXAMPLE

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

VAX Data Types and Instruction Set

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

7 Terminology and Conventions

The following sections describe terminology and conventions used in Part II of this volume.

7.1 Numbering

All numbers, unless otherwise indicated, are decimal. Where there is ambiguity, numbers other than decimal are indicated with the base in English following the number in parentheses. For example:

```
FF (hex)
```

7.2 UNPREDICTABLE and UNDEFINED

Results specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE. Operations specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation might vary from causing no effect to stopping system operation. UNDEFINED operations must not cause the processor to hang—to reach an unhalted state from which there is no transition to a normal state in which the machine executes instructions. Note the distinction between result and operation. Nonprivileged software cannot invoke UNDEFINED operations.

7.3 Ranges and Extents

Ranges are specified in English and are inclusive (for example, a range of integers 0 to 4 includes the integers 0, 1, 2, 3, and 4). Extents are specified by a pair of numbers separated by a colon and are inclusive (that is, bits 7:3 specifies an extent of bits including bits 7, 6, 5, 4, and 3).

7.4 MBZ

Fields specified as MBZ (must be zero) must never be filled by software with a nonzero value. If the processor encounters a nonzero value in a field specified as MBZ, a reserved operand fault or abort occurs if that field is accessible to nonprivileged software. MBZ fields that are accessible only to privileged software (kernel mode) cannot be checked for nonzero value by some or all VAX implementations. Nonzero values in MBZ fields accessible only to privileged software may produce UNDEFINED operation.

7.5 RAZ

Fields specified as RAZ (read as zero) return a zero when read.

7.6 SBZ

Fields specified as SBZ (should be zero) should be filled by software with a zero value. Non-zero values in SBZ fields produce UNPREDICTABLE results and may produce extraneous instruction-issue delays.

7.7 Reserved

Unassigned values of fields are reserved for future use. In many cases, some values are indicated as reserved to CSS and customers. Only these values should be used for nonstandard applications. The values indicated as reserved to Digital and all MBZ (must be zero) fields are to be used only to extend future standard architecture.

7.8 Figure Drawing Conventions

Figures that depict registers or memory follow the convention that increasing addresses extend from right to left and from top to bottom.