PART 6

THE DOS/BATCH ASSEMBLER

MACRO

PART 6

CHAPTER 1

INTRODUCTION TO THE MACRO ASSEMBLER

This chapter presents a brief overview of some fundamental software concepts essential to efficient assembly language programming of the PDP-11 family of computers. A description of the hardware components of the PDP-11 family can be found in two DEC paperback handbooks.

PDP-11 Processor Handbook
PDP-11 Peripherals and Interfacing Handbook

The user is also advised to obtain a PDP-11 Pocket Instruction List card for easy reference. (These items can be obtained from the Software Distribution Center.)

Some notable features of MACRO are:

1. Program and command string control of assembly functions.
2. Device and filename specifications for input and output files.
3. Error listing on command output device.
4. Alphabetized, formatted symbol table listing.
5. Relocatable object modules.
6. Global symbols for linking between object modules.
7. Conditional assembly directives.
8. Program sectioning directives.
9. User-defined macros.
10. Comprehensive set of system macros.
11. Extensive listing control.

No attempt is made in this document to describe the PDP-11 hardware or the function of the various PDP-11 instructions. The reader is advised to become familiar with this material before proceeding.

Assembly language programming deals directly with the host hardware. Therefore great care must be taken in specifying programming standards and conventions, if code written by different groups is to be easily interchanged. When a set of standards guides the entire programming process, the total programming effort becomes easier to plan, comprehend, test, modify, and convert.

The output of a MACRO assembly is a relocatable object module. LINK can bind one or more modules together and create an executable task.

Once built, a program can generally be loaded and executed only at the address specified at LINK time. This is because LINK has had to make adjustments in some codes to reflect the memory locations in which the program is to run.

It is possible to write a source program that can be loaded and run in any section of memory. Such a program consists of position-independent code. The construction of position-independent code is dependent upon the proper usage of PDP-11 addressing modes. (Addressing modes are described in detail in the Processor Handbook.) See Chapter 6-10 for an explanation on how to write position-independent code.

PART 6

CHAPTER 2

SOURCE PROGRAM FORMAT

A source program is composed of a sequence of source lines, where each line contains a single assembly language statement.

An assembly language line can contain up to 132 characters. Beyond this limit an I/O error is generated.

2.1 STATEMENT FORMAT

A statement can contain up to four fields, which are identified by order of appearance and delimited by certain terminating characters. The general format of a MACRO assembly language statement follows.

```
[label:] [operator][operand]   [;comments]
```

The label and comment fields are optional. The operator and operand fields are interdependent; either may be omitted depending upon the contents of the other. However, blank lines are legal.

Some statements have one operand, for example,

```
CLR       RØ
```

while others have two.

```
MOV       #344,R2
```

An assembly language statement must be complete on one source line. No continuation lines are allowed.

MACRO source statements may use the TAB character to align the statement fields according to this standard format:

```
label      - column 1
operator   - column 9
operand(s) - column 17
comments   - column 33
```

For example:

```
1        9        17                  33
REGTST:  BIT      #MASK,VALUE         ;3 BITS?
```

2.1.1 Label Field

A label is a user-defined symbol to which the assembler assigns the value of the current location counter and enters it into the user-defined symbol table. The value of the label may be either absolute or relocatable, depending on whether the location counter value is currently absolute or relocatable. (In the latter case, LINK assigns the absolute value of the symbol by adding the stated relocatable value to the relocation bias calculated by LINK.)

A label is a means of symbolically referring to a specific location within a program. If present, a label always occurs first in a statement and must be terminated by a colon. For example, if the current location is absolute 1ØØ(octal), the assembler processes the statement

```
ABCD:    MOV     A,B
```

and assigns the value 1ØØ(octal) to the label ABCD. Subsequent references to ABCD reference location 1ØØ(octal). In this example if the location counter were relocatable, the assigned value of ABCD would be 1ØØ(octal)+K, where K is the location of the beginning of the relocatable section in which the label ABCD appears.

A double colon defines the label as a global symbol that is accessible to independently assembled modules; the statement

```
ABCD::   MOV     A,B
```

establishes ABCD as a global symbol.

More than one label may appear within a single label field; each label within the field references the same location. For example, if the current location counter is 1ØØ(octal), the multiple labels in the statement

```
ABC:    $DD:    A7.7:    MOV     A,B
```

causes each of the three labels ABC, $DD, and A7.7 to be assigned the value 1ØØ(octal).

The legal label characters are

```
A - Z
Ø - 9
  .
  $
```

(By convention, $ and . characters are reserved for use in system software symbols.)

The first six characters of a label are significant. An error message is generated if two or more labels share the same first six characters.

A symbol used as a label may not be redefined within the user program. An attempt to redefine a label results in an error flag (M) in the assembly listing.

2.1.2 Operator Field

An operator field follows the label field in a statement, and may contain a macro call, an instruction mnemonic, or an assembler directive. The operator may be preceded by none, one or more labels and may be followed by none, one or more operands and/or a comment. Leading and trailing spaces and tab characters are ignored.

When the operator is a macro call, the assembler inserts the appropriate code to expand the macro. When the operator is an instruction mnemonic, it specifies the instruction to be generated and the action to be performed on any operand(s) that follow. When the operator is an assembler directive, it specifies a certain function or action to be performed during assembly.

An operator is legally terminated by a space, tab, or any nonalphanumeric character.

Consider the following examples.

```
MOV   A,B      ;space terminates the operator MOV
MOV@A,B        ;@ terminates the operator MOV
```

A blank operator field is interpreted as a .WORD assembler directive (see Section 6-5.3.2).

2.1.3 Operand Field

An operand is that part of a statement that is manipulated by the operator. Operands may be expressions, numbers, symbolic names, or macro arguments (within the context of the operation). When multiple operands appear within a statement, each is separated from the next by one of the following characters: comma, tab, space, or paired angle brackets around one or more operands (see Section 6-3.1.1). An operand may be preceded by an operator, label, or other operand and followed by another operand or a comment.

The operand field is terminated by a semicolon when followed by a comment, or by a statement terminator when the operand completes the statement. For example:

```
LABEL:  MOV     A,B          ;COMMENT
```

The tab between MOV and A terminates the operator field and begins the operand field; a comma separates the operands A and B; a semicolon terminates the operand field and begins the comment field.

2.1.4 Comment Field

The comment field is optional and may contain any ASCII characters except null, rubout, carriage return, line feed, vertical tab or form feed. All other characters, even special characters with a defined use, are ignored by the assembler when appearing in the comment field.

The comment field may be preceded by any or none of the other three field types. Comments must begin with the semicolon character.

Comments do not affect assembly processing or program execution, but are useful in source listings for later analysis, debugging, or documentation purposes.

2.2 FORMAT CONTROL

Horizontal or line formatting of the source program is controlled by the space and tab characters. These characters have no effect on the assembly process unless they are embedded within a symbol, number, or ASCII text; or unless they are used as the operator field terminator. Thus, these characters can be used to provide an orderly source program.

```
LABEL:   MOV    (SP)+,TAG          ;POP VALUE OFF STACK
```

(See Section 6-5.1.2 for a description of page formatting with respect to macros, and Section 6-5.1.1 for a description of assembly listing output.)

PART 6

CHAPTER 3

SYMBOLS AND EXPRESSIONS

This chapter describes the various components of legal MACRO expressions, the assembler character set, symbol construction, numbers, operators, terms, and expressions.

3.1 CHARACTER SET

The following characters are legal in MACRO source programs:

1. The letters A through Z. Both upper and lower case letters are acceptable; although, upon input, lower case letters are converted to upper case letters. Lower case letters can only be output by sending their ASCII values to the output device. This conversion is not true for .ASCII, .ASCIZ, ' (single quote) or " (double quote) statements if .ENABL LC is in effect.

2. The digits Ø through 9.

3. The characters . (period or dot) and $ (dollar sign), are reserved for use in system program symbols.

4. The special characters in Table 6-1.

Table 6-1
MACRO Special Characters

Character	Designation	Function
:: ==	double colon double equal sign	Either the double colon or double equal sign may be used to define a symbol as a global symbol (refer to Section 6-3.5).
:	colon	label terminator
=	equal sign	direct assignment indicator
%	percent sign	register term indicator
	tab	item or field terminator
	space	item or field terminator
#	number sign	immediate expression indicator
@	at sign	deferred addressing indicator
(	left parenthesis	initial register indicator
)	right parenthesis	terminal register indicator
,	comma	operand field separator

(continued on next page)

Table 6-1 (Cont.)
MACRO Special Characters

Character	Designation	Function
;	semicolon	comment field indicator
<	left angle bracket	initial argument or expression indicator
>	right angle bracket	terminal argument or expression indicator
+	plus sign	arithmetic addition operator or auto-increment indicator
-	minus sign	arithmetic subtraction operator or auto-decrement indicator
*	asterisk	arithmetic multiplication operator
/	slash	arithmetic division operator
&	ampersand	logical AND operator
!	exclamation	logical inclusive OR operator
"	double quote	double ASCII character indicator
'	single quote	single ASCII character indicator
↑ ^	up arrow or circumflex	universal unary operator, argument indicator
\	backslash	macro numeric argument indicator

3.1.1 Separating and Delimiting Characters

Reference is made in the remainder of the manual to legal separating characters and legal argument delimiters. These terms are defined in Tables 6-2 and 6-3.

Table 6-2
Legal Separating Characters

Character	Definition	Usage
space	one or more spaces and/or tabs	A space is a legal separator only for argument operands. Spaces within expressions are ignored (see Section 6-3.3).
,	comma	A comma is a legal separator for both expressions and arguments.

Table 6-3
Legal Delimiting Characters

Character	Definition	Usage
<...>	paired angle brackets	Paired angle brackets are used to enclose an argument, particularly when that argument contains separating characters. Paired angle brackets may be used anywhere in a program to enclose an expression for treatment as a term.
↑/.../	up arrow construction where the up arrow character is followed by an argument bracketed by any paired printing characters	This construction is equivalent in function to the paired angle brackets and is generally used only where the argument contains angle brackets.

Where argument delimiting characters are used, they must bracket the first (and, optionally, any following) argument(s). The character < and the characters ↑x, where x is any printing character, can be considered unary operators that cannot be immediately preceded by another argument. For example

```
.MACRO     TEM <AB>C
```

indicates a macro definition with two arguments, while

```
.MACRO     TEL C<AB>
```

has only one argument. The closing character (or matching character where the up arrow construction is used) acts as a separator. The opening argument delimiter does not act as an argument separator.

Angle brackets can be nested as follows:

```
D,<A<B>C>
```

which reduces to:

```
D,A<B>C
```

and which is considered to be two arguments in both forms.

3.1.2 Illegal Characters

A character can be illegal in one of two ways:

1. A character that is not recognized as an element of the MACRO character set is always an illegal character and causes immediate termination of the current line at that point and an error flag (I) in the assembly listing. For example in the statement

```
LABEL←*A:   MOV  A,B
```

   the backarrow is not a recognized character. The entire line is treated as

```
.WORD   LABEL
```

   and is flagged in the listing.

2. A legal MACRO character may be illegal in context. Such a character generates a Q error on the assembly listing.

3.1.3 Operators

Table 6-4 shows legal unary operators under MACRO.

Table 6-4
MACRO Unary Operators

Unary Operator	Explanation
+	plus sign, positive value
-	minus sign, negative 2's complement value
↑	up arrow, universal unary operator

The unary operators described in Table 6-4 can be used adjacent to each other in a term.

Table 6-5 shows legal binary operators under MACRO.

Table 6-5
MACRO Binary Operators

Binary Operator	Explanation
+	addition
-	subtraction
*	multiplication
/	division
&	logical AND
!	logical inclusive OR

All binary operators have the same priority. Items can be grouped for evaluation within an expression by enclosure in angle brackets. Terms in angle brackets are evaluated first, and remaining operations are performed left to right. See the following examples.

```
.WORD    1+2*3        ;IS 11 OCTAL
.WORD    1+<2*3>      ;IS 7 OCTAL
```

3.2 TERMS

A term is a component of an expression. A term may be one of the following.

1. A number, as defined in Section 6-3.9.
2. A symbol, as defined in Section 6-3.4.
3. An ASCII conversion as defined in Section 6-5.3.3.
4. An expression or term enclosed in angle brackets. Any quantity enclosed in angle brackets is evaluated before the remainder of the expression in which it is found. Angle brackets are used to alter the left-to-right evaluation of expressions (to differentiate between A*B+C and A*<B+C>) or to apply a unary operator to an entire expression (-<A+B>, for example).

3.3 EXPRESSIONS

Expressions are combinations of terms and operators that reduce to a 16-bit value. The evaluation of an expression includes the evaluation of the mode of the resultant expression; that is, absolute, relocatable or external.

Expressions are evaluated left to right with no operator hierarchy rules except that unary operators take precedence over binary operators. A term preceded by a unary operator can be considered as containing that unary operator. (Terms are evaluated, where necessary, before their use in expressions.) Multiple unary operators are valid and are treated as follows:

```
-+-A
```

is equivalent to

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

A missing term, expression, or external symbol is interpreted as a zero. A missing operator terminates the expression analysis. A Q error flag is generated for each missing term or operator. For example:

```
TAG!LA 177777
```

is evaluated as

```
TAG!LA
```

with a Q error flag on the assembly listing line.

The value of an external expression is the value of the absolute part of the expression; e.g., EXTERNAL+A has a value of A. This is modified by LINK to become EXTERNAL+A.

Expressions, when evaluated, are either absolute, relocatable, or external. For the programmer writing position-independent code, the distinction is important.

1. An expression is absolute if its value is fixed. An expression whose terms are numbers and ASCII conversions will have an absolute value. A relocatable expression minus a relocatable term, where both items belong to the same program section, is also absolute.

2. An expression is relocatable if its value is fixed relative to a base address but will have an offset value added at link time. Expressions whose terms contain labels defined in relocatable sections and periods (in relocatable sections) will have a relocatable value.

3. An external expression is one whose partial definition at assembly time is completed at linking time. Also, an external expression is one whose terms may contain global symbols not defined in the current program. At linking time, external expressions containing relocatable global symbols are considered relocatable; those containing absolute globals are considered absolute.

3.4 MACRO SYMBOLS

There are three types of symbols: permanent, user-defined, and macro. MACRO maintains three types of symbol tables; the permanent symbol table (PST), the user symbol table (UST), and the macro symbol table (MST). The PST contains all the permanent symbols and is part of the MACRO Assembler load module. The UST and MST are constructed as the source program is assembled. User-defined symbols are added to the table as they are encountered.

Symbols are interpreted according to the following hierarchy:

a. A period causes the value of the current location counter to be used.

b. A permanent symbol's basic value is used, but its arguments (if any) are ignored.

c. An undefined symbol is assigned a value of zero and is inserted in the user-defined symbol table as an undefined global reference. If the .DSABL GBL directive is in effect, the automatic global reference default function is inhibited, and the symbol is not defined as a global reference. It remains undefined. Refer to Section 6-5.2.

3.4.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics and assembler directives (see Chapter 6-5, 6-6, and 6-8). These symbols are primitives of the assembler and need not be defined before being used in the source program.

3.4.2 User-Defined and Macro Symbols

User-defined symbols are those used as labels (Section 6-2.1.1) or defined by direct assignment (Section 6-3.5). These symbols are added to the user symbol table as they are encountered during the first pass of the assembly. Macro symbols are those symbols used as macro names (Section 6-6.1). These symbols are added to the macro symbol table as they are encountered during the assembly.

User-defined and macro symbols can be composed of alphanumeric characters, dollar signs, and periods only; any other character is illegal.

The $ and . are in general use by system software, and the user is advised to avoid their use.

The following rules apply to the creation of user-defined and macro symbols:

1. The first character must not be a number (except in the case of local symbols, see Section 6-3.7).

2. Each symbol must be unique within the first six characters.

3. A symbol can be written with more than six legal characters, but the seventh and subsequent characters are only checked for legality, and are not otherwise recognized by the assembler.

4. Spaces, tabs, and illegal characters must not be embedded within a symbol.

The value of a symbol depends upon its use in the program. A symbol in the operator field may be any one of the three symbol types. To determine the value of the symbol, the assembler searches the three symbol tables in the order Macro Symbol Table, Permanent Symbol Table, User Symbol Table.

A symbol found in the operand field is sought in the order User-Defined Symbol Table and Permanent Symbol Table.

These search orders allow redefinition of permanent symbol table entries as user-defined or macro symbols. The same name can also be assigned to both a macro and a label.

User-defined symbols are either internal or external (global). All user-defined symbols are internal unless they remain undefined internally or unless explicitly defined as being global with the .GLOBL directive or by the double-colon, or double-equal sign (see Section 6-5.10).

Global symbols provide links between object modules. A global symbol that is defined as a label is generally called an entry point (to a section of code). Such symbols are referenced from other object modules to transfer control throughout the load module (which may be composed of a number of object modules).

Since MACRO provides program sectioning capabilities (Section 6-5.9), two types of internal symbols must be considered: symbols that belong to the current program section, and symbols that belong to other program sections. In both cases, the internal symbol must be defined within the current assembly; this is critical in evaluating expressions involving the second type of internal symbol (see Section 6-3.3).

3.5 DIRECT ASSIGNMENT

A direct assignment statement associates a symbol with a value. When a direct assignment statement defines a symbol for the first time, that symbol is entered into the user symbol table (UST). A symbol may be redefined by assigning a new value to a previously defined symbol. The latest assigned value replaces any previous value assigned to a symbol.

The general format for a direct assignment statement follows:

```
sym = expression

or

sym == expression
```

The second statement also defines sym as a global symbol.

Symbols take on the relocatable or absolute attribute of their defining expression. However, if the defining expression is external, the symbol is not global unless explicitly defined as such in a .GLOBL directive, by a label delimited by a double colon, or by the double equal sign (see Section 6-5.10). Global references in an expression assigned to a symbol are illegal, and are flagged with an A error flag.

The following conventions apply to direct assignment statements.

1. An equal sign (=) or double equal (==) must separate the symbol from the expression defining the symbol value.
2. A direct assignment statement is usually placed in the label field and may be followed by a comment.
3. Only one symbol can be defined in a single direct assignment statement.
4. Only one level of forward referencing is allowed.

Example of two levels of forward referencing (illegal):

```
X = Y
Y = Z
Z = 1
```

3.6 REGISTER SYMBOLS

The eight general registers of the PDP-11 are numbered Ø through 7 and can be expressed in the source program as

```
%Ø
%1
.
.
.
%7
```

where the digit indicating the specific register can be replaced by any legal term that can be evaluated during the first assembly pass.

It is recommended that the programmer use symbolic names for all register references. Unless the .DSABL REG statement has been encountered, the definitions as shown in the following example are defined by default; alternatively, a register symbol may be defined in a direct assignment statement among the first statements in the program. The defining expression of a register symbol must be absolute. For example:

```
RØ=%Ø              ;REGISTER DEFINITION
R1=%1
R2=%2
R3=%3
R4=%4
R5=%5
SP=%6
PC=%7
```

The user can reassign the register expressions, if he wishes.

The symbolic names assigned to the registers in the example above are the conventional names used in all PDP-11 system programs. Since these names are mnemonic, it is suggested the user follow this convention. Note that registers 6 and 7 are given special names because of their special functions.

All register symbols must be defined before they are referenced. A forward reference to a register symbol is flagged as an error.

The % character may be used with any term or expression to specify a register. (A register expression less than Ø or greater than 7 is flagged with an R error code.) For example, the statement

```
CLR   %3+1
```

is equivalent to

```
CLR   %4
```

and clears the contents of register 4; while

```
CLR   4
```

clears the contents of memory address 4.

3.7 LOCAL SYMBOLS

Local symbols are specially formatted symbols used as labels within a given range.

Local symbols provide a convenient means of generating labels for branch instruction, etc. Use of local symbols reduces the possibility of multi-defined symbols within a user program and separates entry point symbols from local references. Local symbols may not be referenced from other object modules or even from outside their local symbol block. The rules for delimiting a local symbol block appear below.

Local symbols are of the form n$ where n is a decimal integer from 1 to 65535, inclusive, and can only be used on word boundaries (i.e., at even addresses). Local symbols include the following.

```
  1$
 27$
 59$
1Ø4$
```

Within a local symbol block, local symbols can be defined and referenced. However, a local symbol cannot be referenced outside the block in which it is defined. There is no conflict with labels of the same name in other local symbol blocks.

Local symbols 64$ through 127$ can be generated automatically as a feature of the macro processor (see Section 6-6.3.6 for further details). When using local symbols, the user is advised to first use the range from 1$ to 63$, or the range from 128$ to 65535$.

A local symbol block is delimited in one of the following ways:

1. The range of a single local symbol block can consist of those statements between two normally constructed symbolic labels. (Note that a statement of the form

   LABEL=.

   is a direct assignment; it does not create a label in the strict sense, and does not delimit a local range.)

2. The range of a local symbol block is always terminated upon encountering a .PSECT, .CSECT, or .ASECT directive.

3. The range of a single local symbol block can be delimited with .ENABL LSB and the first symbolic label or .PSECT, .CSECT, or .ASECT directive following .DSABL LSB directive.

For examples of local symbols and local symbol blocks, see Figure 6-1.

```
 Line      Octal
Number    Expansion      Source Code                          Comments
  1                      .SBTTL   SECTOR INITIALIZATION
  2
  3       000000'        .CSECT   IMPURE              ;IMPURE STORAGE AREA
  4 000000         IMPURE:
  5       000000'        .CSECT   IMPPAS              ;CLEARED EACH PASS
  6 000000         IMPPAS:
  7       000000'        .CSECT   IMPLIN              ;CLEARED EACH LINE
  8 000000         IMPLIN:
  9
 10       000000'        .CSECT   XCTPRG              ;PROGRAM INITIALIZATION CODE
 11 00000         XCTPRG:
 12 00000 012700        MOV      #IMPURE,R0
          000000'
 13 00004 005020  1$:    CLR      (R0)+               ;CLEAR IMPURE AREA
 14 00006 022700        CMP      #IMPTOP,R0
          00004'
 15 00012 101374        BHI      1$
 16
 17       000000'        .CSECT   XCTPAS              ;PASS INITIALIZATION CODE
 18 00000         XCTPAS:
 19 00000 012700        MOV      #IMPPAS,R0
          000000'
 20 00004 005020  1$:    CLR      (R0)+               ;CLEAR IMPURE PART
 21 00006 022700        CMP      #IMPTOP,R0
          00004'
 22 00012 101374        BHI      1$
 23
 24       000000'        .CSECT   XCTLIN              ;LINE INITIALIZATION CODE
 25 00000         XCTLIN:
 26 00000 012700        MOV      #IMPLIN,R0
          000000'
 27 00004 005020  1$:    CLR      (R0)+
 28 00006 022700        CMP      #IMPTOP,R0
          00004'
 29 00012 101374        BHI      1$
 30
 31       000000'        .CSECT   MIXED               ;MIXED MODE SECTOR
```

Figure 6-1
Assembly Source Listing of MACRO Code Showing Local Symbol Blocks

3.8 ASSEMBLY LOCATION COUNTER

The period (.) is the symbol for the assembly location counter. When used in the operand field of an instruction, it represents the address of the first word of the instruction. When used in the operand field of an assembler directive, it represents the address of the current byte or word. For example:

```
A:      MOV     #.,RØ           ;. REFERS TO LOCATION A,
                                ;I.E., THE ADDRESS OF THE
                                ;MOV INSTRUCTION.
```

(# is explained in Section 6-4.1.)

At the beginning of each assembly pass, the assembler clears the location counter. Normally, consecutive memory locations are assigned to each byte of object data generated. However, the location where the object data is stored may be changed by a direct assignment altering the location counter.

Example:

```
.=expression
```

The location counter symbol has a mode associated with it, either absolute or relocatable. The existing mode of the location counter cannot be changed by using a defining expression of a different mode.

The mode of the location counter symbol can be changed by the use of the .ASECT, .CSECT or .PSECT directives as explained in Section 6-5.9.2.

The expression defining the location counter must not contain forward references or symbols that vary from one pass to another.

Examples:

```
        .ASECT

.=5ØØ                           ;SET LOCATION COUNTER
                                ;ABSOLUTE 5ØØ

FIRST:  MOV     .+1Ø,COUNTY     ;THE LABEL FIRST HAS THE VALUE
                                ;5ØØ(OCTAL)
                                ;.+1Ø EQUALS 51Ø(OCTAL). THE
                                ;CONTENTS OF THE LOCATION
                                ;51Ø(OCTAL) WILL BE DEPOSITED
                                ;IN LOCATION COUNTY.

.=52Ø                           ;THE ASSEMBLY LOCATION COUNTER
                                ;NOW HAS A VALUE OF
                                ;ABSOLUTE 52Ø(OCTAL)
```

```
SECOND: MOV       .,INDEX          ;THE LABEL SECOND HAS THE
                                   ;VALUE 52Ø(OCTAL)
                                   ;THE CONTENTS OF LOCATION
                                   ;52Ø(OCTAL), THAT IS, THE BINARY
                                   ;CODE FOR THE INSTRUCTION
                                   ;ITSELF, WILL BE DEPOSITED IN
                                   ;LOCATION INDEX.

        .PSECT

.=.+2Ø                             ;SET LOCATION COUNTER TO
                                   ;RELOCATABLE 2Ø OF THE
                                   ;UNNAMED PROGRAM SECTION.

THIRD:  .WORD     Ø                ;THE LABEL THIRD HAS THE
                                   ;VALUE OF RELOCATABLE 2Ø.
```

Storage area may be reserved by advancing the location counter. For example, if the current value of the location counter is 1ØØØ, the direct assignment statements

```
.=.+4Ø
; or
        .BLKB 4Ø
; or
        .BLKW 2Ø
```

reserve 4Ø(octal) bytes of storage space in the program. The next instruction is stored at 11ØØ. (The .BLKB and .BLKW directives are recommended as the preferred ways to reserve space. Refer to Section 6-5.5.3.)

3.9 NUMBERS

The MACRO Assembler assumes all numbers in the source program are to be interpreted in octal radix unless otherwise specified. The assumed radix can be altered with the .RADIX directive (see Section 6-5.4.1) or individual numbers can be treated as being of decimal, binary, or octal radix (see Section 6-5.4.2).

Octal numbers consist of the digits Ø through 7 only. A number not specified as a decimal number and containing an 8 or 9 is flagged with an N error code and treated as a decimal number.

Negative numbers are preceded by a minus sign (the assembler translates them into two's complement form). Positive numbers may be preceded by a plus sign, although this is not required.

A number that does not fit into 16 bits (n>177777) is truncated from the left and flagged with a T error code in the assembly listing.

Numbers are always considered absolute quantities (that is, not relocatable).

Single-word floating-point numbers may be generated with the ↑F operator (see Section 6-5.6.2). Refer to PDP-11 Processor Handbook for details of the floating-point format.

3.10 RELOCATION AND LINKING

The output of the MACRO Assembler is an object module that must be processed by LINK before loading and execution. (See Part 9 of this manual for details.) LINK essentially fixes (i.e., makes absolute) the values of external or relocatable symbols and turns the object module into a load module.

To enable the Linker to determine the value of an expression, the assembler issues certain directives to LINK, together with required parameters. In the case of relocatable expressions, LINK adds the base of the associated relocatable section (the location in memory of relocatable Ø) to the value of the relocatable expression provided by the assembler. In the case of an external expression, LINK determines the value of the external term in the expression (since the external symbol must be defined in one of the other object modules which are being linked together), and adds it to the value of the external expression provided by the assembler.

All instructions that are to be modified (as described in the previous paragraph) are marked with an apostrophe in the assembly listing (see also Chapter 6-10). Thus, the binary text output looks like the following.

```
ØØ5Ø65   CLR     EXTERNAL(5)
ØØØØØØ'                              ;VALUE OF EXTERNAL SYMBOL
                                     ;ASSEMBLED ZERO; WILL BE
                                     ;MODIFIED BY LINK.

ØØ5Ø65   CLR     EXTERNAL+6(5)       ;THE ABSOLUTE PORTION OF THE
ØØØØØ6'                              ;EXPRESSION (ØØØØØ6) IS ADDED
                                     ;BY LINK TO THE VALUE
                                     ;OF THE EXTERNAL SYMBOL

ØØ5Ø65   CLR     RELOCATABLE(5)      ;ASSUMING WE ARE IN A
                                     ;RELOCATABLE
ØØØØ4Ø'                              ;SECTION AND THE VALUE OF
                                     ;RELOCATABLE SYMBOL IS RELOCATABLE 4Ø
                                     ;LINK WILL ADD
                                     ;THE RELOCATION BIAS TO 4Ø
```

PART 6

CHAPTER 4

ADDRESSING INFORMATION

Please refer to the PDP-11 Processor Handbook for complete information and examples concerning addressing modes. This chapter serves only to summarize that information.

4.1 MODE FORMS AND CODES

Each instruction takes at least one word. Operands of the forms listed in Table 6-6 do not increase the length of an instruction.

Table 6-6
Address Modes - No Instruction Modification

Op Code	Operand	Mode	Meaning
op	R	Øn	Register mode
op	@R or (ER)	1n	Register deferred mode
op	(ER)+	2n	Autoincrement mode
op	@(ER)+	3n	Autoincrement deferred mode
op	-(ER)	4n	Autodecrement mode
op	@-(ER)	5n	Autodecrement deferred mode

n is the register number.

However, any of the forms in Table 6-7 adds one word to the instruction length.

Table 6-7
Address Modes - Instruction Modifying

Op Code	Operand	Mode	Meaning
op	E (ER)	6n	Index mode
op	@E(ER)	7n	Index deferred mode
op	#E	27	Immediate mode
op	@#E	37	Absolute memory reference mode
op	E	67	Relative mode
op	@E	77	Relative deferred reference mode

n is the register number. Note that in the last four forms, register 7 (the PC) is referenced.

NOTE

An alternate form for @R is (ER). However, the form @(ER) is equivalent to @Ø(ER).

The form @#E differs from the form E in that the second or third word of the instruction contains the absolute address of the operand rather than the relative distance between the operand and the

PC. Thus, the instruction CLR @#1ØØ clears absolute location 1ØØ even if the instruction is moved from the point at which it was assembled. See the description of the .ENABLE AMA function in Section 6-5.2, which directs the assembly of all relative mode addresses as absolute mode addresses.

4.2 BRANCH INSTRUCTION ADDRESSING

The branch instructions are 1-word instructions. The high byte contains the op-code and the low byte contains an 8-bit signed word offset (seven bits plus sign) that specifies the branch address relative to the PC. The hardware calculates the branch address as follows:

1. Extend the sign of the word offset through bits 8-15.
2. Multiply the result by 2. This creates a byte offset from a word offset.
3. Add the result to the PC to form the final branch address.

The assembler performs the reverse operation to form the word offset from the specified byte address, when assembling the instruction. Remember that when the byte offset is added to the PC, the PC is pointing to the word following the branch instruction; hence the factor -2 in the following calculation.

word offset = (E-PC)/2 truncated to eight bits.

Since PC = .+2, we have

word offset = (E-.-2)/2 truncated to eight bits.

NOTE

It is illegal to branch to a location specified as an external symbol, to a relocatable symbol from within an absolute section, or to an absolute or relocatable symbol or another program section from within a relocatable section.

The EMT and TRAP instructions use the low-order byte of the instruction word for user-defined codes. This allows information to be transferred to the trap handlers via this low-order byte. If EMT or TRAP is followed by an expression, the value is put into the low-order byte of the word. However, if the expression is too big (>377(octal)) it is truncated to eight bits and a T error flag is generated.

PART 6

CHAPTER 5

GENERAL ASSEMBLER DIRECTIVES

5.1 LISTING CONTROL DIRECTIVES

5.1.1 .LIST and .NLIST

Listing options can be specified in the text of a MACRO program through the .LIST and .NLIST directives. These are of the form

```
.LIST   [arg]
.NLIST  [arg]
```

where arg represents one or more optional arguments.

When used without arguments, the listing directives alter the listing level count. The listing level count causes the listing to be suppressed when it is negative. The count is initialized to zero, incremented for each .LIST and decremented for each .NLIST. For example:

```
        .MACRO LTEST                    ;LIST TEST
; A-THIS LINE SHOULD LIST
        .NLIST
; B-THIS LINE SHOULD NOT LIST
        .NLIST
; C-THIS LINE SHOULD NOT LIST
        .LIST
; D-THIS LINE SHOULD NOT LIST (LEVEL NOT BACK TO ZERO)
        .LIST
; E-THIS LINE SHOULD LIST (LEVEL BACK TO ZERO)
        .ENDM

        LTEST                           ;CALL THE MACRO

; A-THIS LINE SHOULD LIST
        .NLIST
        .LIST
; E-THIS LIST SHOULD LIST (LEVEL BACK TO ZERO)
```

The primary purpose of the level count is to allow macro expansions to be selectively listed and yet exit with the level returned to the status current during the macro call.

The use of arguments with the listing directives does not affect the level count; however, use of .LIST and .NLIST can be used to override the current listing control.

For example:

```
        .MACRO XX
          .
          .
          .
        .LIST               ;LIST NEXT LINE
X=.
        .NLIST              ;DO NOT LIST REMAINDER
          .
          .                 ;OF MACRO EXPANSION
          .
        .ENDM
        .NLIST ME           ;DO NOT LIST MACRO EXPANSIONS
        XX
        .LIST               ;LIST NEXT LINE
X=.
```

Allowable arguments for use with the listing directives appear in Table 6-8. These arguments can be used singly or in combination.

Table 6-8
MACRO Listing Directive Arguments

Argument	Default	Function
SEQ	list	Controls the listing of source line sequence numbers. Error flags are normally printed on the line preceding the questionable source statement.
LOC	list	Controls the listing of the location counter (this field would not normally be suppressed).
BIN	list	Controls the listing of generated binary code.
BEX	list	Controls listing of binary extensions; that is, those locations and binary contents beyond the first binary word (per source statement). This is a subset of the BIN argument.
SRC	list	Controls the listing of the source code.
COM	list	Controls the listing of comments. This is a subset of the SRC argument and can be used to reduce listing time and/or space where comments are unnecessary.
MD	list	Controls listing of macro definitions and repeat range expansions.
MC	list	Controls listing of macro calls and repeat range expansions.
ME	no list	Controls listing of macro expansions.
MEB	no list	Controls listing of macro expansion binary code. A LIST MEB causes only those macro expansion statements producing binary code to be listed. This is a subset of the ME argument.

(continued on next page)

Table 6-8 (cont.)
MACRO Listing Directive Arguments

Argument	Default	Function
CND	list	Controls the listing of unsatisfied conditions and all .IF and .ENDC statements. This argument permits conditional assemblies to be listed without including unsatisfied code.
LD	no list	Control listing of all listing directives having no arguments (those used to alter the listing level count).
TOC	list	Control listing of table of contents on pass 1 of the assembly (see Section 6-5.1.4 describing the .SBTTL directive). The full assembly listing is printed during pass 1 of the assembly.
TTM	console mode	Control listing output format. The TTM argument (the default case) causes output lines to be truncated to 72 characters. Binary code is printed with the binary extensions below the first binary word. The alternative (.NLIST TTM) to terminal mode is line printer mode, which is shown in Figure 6-2.
SYM	list	Controls the listing of the symbol table for the assembly.

An example of an assembly listing as sent to a 132-column line printer is shown in Figure 6-2. Notice that binary extensions for statements generating more than one word are spread horizontally on the source line. An example of an assembly listing as sent to a teleprinter is shown in Figure 6-3. Notice that binary extensions for statements generating more than one word are printed on subsequent lines.

The listing options can also be specified through switches on the listing file specification in the command string to the MACRO Assembler. These switches are

```
/LI:arg
/NL:arg
```

where arg is any one or more of the arguments defined in the .LIST and .NLIST directive.

```
 1 ØØ1766                                GETLIN:                         ;GET AN INPUT LINE
 2 ØØ1766                                       SAVREG
 3 ØØ1772  Ø167ØØ  ØØØØ2Ø'              1$:     MOV     FFCNT,RØ        ;ANY RESERVED FF'S?
 4 ØØ1776  ØØ142Ø                               BEQ     31$             ;  NO
 5 ØØ2ØØØ  Ø6ØØ67  ØØØØ22'                     ADD     RØ,PAGNUM       ;YES, UPDATE PAGE NUMBER
 6 ØØ2ØØ4  Ø12767  177777  ØØØØ26'             MOV     #-1,PAGEXT
 7 ØØ2Ø12  ØØ5Ø67  ØØØØ12'                     CLR     LINNUM          ;INIT NEW CREF SEQUENCE
 8 ØØ2Ø16  ØØ5Ø67  ØØØØ2Ø'                     CLR     FFCNT
 9 ØØ2Ø22  ØØ5Ø67  ØØØØ16'                     CLR     SEQEND
1Ø ØØ2Ø26  ØØ5767  ØØØØØØ'                     TST     PASS
11 ØØ2Ø32  ØØ14Ø2                              BEQ     31$
12 ØØ2Ø34  ØØ5Ø67  ØØØØ1Ø'                     CLR     LPPCNT
13 ØØ2Ø4Ø  Ø127Ø2  ØØ1712'              31$     MOV     #LINBUF,R2
14 ØØ2Ø44  Ø1Ø267  ØØØØ12'                     MOV     R2,LCBEGL       ;SEAT UP BEGINNING
15 ØØ2Ø5Ø  Ø12767  ØØ2116' ØØØØ14'             MOV     #LINEND,LCENDL  ;  AND END OF LINE MARKERS
17 ØØ2Ø56  ØØ5767  ØØØ2ØØ'                     TST     SMLCNT          ;IN SYSTEM MACRO?
18 ØØ2Ø62  ØØ1145                              BNE     4Ø$             ;  YES, SPECIAL
21 ØØ2Ø64  Ø167Ø1  ØØ2214'                     MOV     MSBMRP,R1       ;ASSUME MACRO IN PROGRESS
22 ØØ2Ø7Ø  ØØ1166                              BNE     1Ø$             ;BRANCH IF SO
24 ØØ2Ø72  Ø127Ø1  ØØØ756'                     MOV     #SRCBUF,R1
25 ØØ2Ø76                                      .WAIT   #SRCLNK
26 ØØ21Ø4  ØØ5267  ØØØØ12'                     INC     LINNUM
27 ØØ211Ø  1167ØØ  ØØØ753'                     MOVB    SRCHDR+3,RØ     ;GET CODE BYTE
28 ØØ2114  Ø327ØØ  ØØØØ47                      BIT     #Ø47,RØ         ;ANYTHING BAD?
29 ØØ212Ø  ØØ14Ø3                              BEQ     32$             ;  NO
3Ø ØØ2122                                      ERROR   L               ;YES, ERROR
31 ØØ213Ø  1Ø61ØØ                      32$:    ROLB    RØ              ;EOF?
32 ØØ2132  1ØØØ14                              BPL     2$              ;  NO
33 ØØ2134  Ø56767  ØØØØØ6' ØØØØØ4'             BIS     CSISAV,ENDFLG
34 ØØ2142  ØØ1ØØ3                              BNE     34$
```

Figure 6-2
Example of MACRO Line Printer Listing
(132 column line printer)

```
1 ØØ1766          GETLIN:                              ;GET AN INPUT LINE
2 ØØ1766                  SAVREG
3 ØØ1772 Ø167ØØ 1$:       MOV     FFCNT,RØ             ;ANY RESERVED FF'S?
         ØØØØ2Ø'
4 ØØ1776 ØØ142Ø           BEQ     31$                  ;  NO
5 ØØ2ØØØ Ø6ØØ67           ADD     RØ,PAGNUM            ;YES, UPDATE PAGE NUMBER
         ØØØØ22'
6 ØØ2ØØ4 Ø12767           MOV     #-1,PAGEXT
         177777
         ØØØØ26'
7 ØØ2Ø12 ØØ5Ø67           CLR     LINNUM               ;INIT NEW CREF SEQUENCE
         ØØØØ12'
8 ØØ2Ø16 ØØ5Ø67           CLR     FFCNT
         ØØØØ2Ø'
9 ØØ2Ø22 ØØ5Ø67           CLR     SEQEND
         ØØØØ16'
1Ø Ø2Ø26 ØØ5767           TST     PASS
         ØØØØØØ'
11 Ø2Ø32 ØØ14Ø2           BEQ     31$
12 Ø2Ø34 ØØ5Ø67           CLR     LPPCNT
         ØØØØ1Ø'
13 Ø2Ø4Ø Ø127Ø2 31$:      MOV     #LINBUF,R2
         ØØ1712'
14 Ø2Ø44 Ø1Ø267           MOV     R2,LCBEGL            ;SEAT UP BEGINNING
         ØØØØ12'
15 Ø2Ø5Ø Ø12767           MOV     #LINEND,LCENDL       ;  AND END OF LINE MARKERS
         ØØ2116'
         ØØØØ14'
16                        .IF NDF XSML
17 Ø2Ø56 ØØ5767           TST     SMLCNT               ;IN SYSTEM MACRO?
         ØØØ2ØØ'
18 Ø2Ø62 ØØ1145           BNE     4Ø$                  ;  YES, SPECIAL
19                        .ENDC
2Ø                        .IF NDF XMACRO
21 Ø2Ø64 Ø167Ø1           MOV     MSBMRP,R1            ;ASSUME MACRO IN PROGRESS
         ØØ2214'
22 Ø2Ø7Ø ØØ1166           BNE     1Ø$                  ;BRANCH IF SO
23                        .IFTF
24 Ø2Ø72 Ø127Ø1           MOV     #SRCBUF,R1
         ØØØ756'
25 Ø2Ø76                  .WAIT   #SRCLNK
26 Ø21Ø4 ØØ5267           INC     LINNUM
         ØØØØ12'
27 Ø211Ø 1167ØØ           MOVB    SRCHDR+3,RØ          ;GET CODE BYTE
         ØØØ753'
28 Ø2114 Ø327ØØ           BIT     #Ø47,RØ              ;ANYTHING BAD?
         ØØØØ47
29 Ø212Ø ØØ14Ø3           BEQ     32$                  ;  NO
3Ø Ø2122                  ERROR   L                    ;YES, ERROR
31 Ø213Ø 1Ø61ØØ 32$:      ROLB    RØ                   ;EOF?
32 Ø2132 1ØØØ14           BPL     2$                   ;  NO
33 Ø2134 Ø56767           BIS     CS1SAV,ENDFLG
         ØØØØØ6'
         ØØØØØ4'
34 Ø2142 ØØ1ØØ3           BNE     34$
```

Figure 6-3
Example of Page Heading from MACRO Teleprinter Listing
(same format as for 8Ø column line printer)

NOTE

Where no listing file specification is indicated, any errors encountered are printed on the teleprinter. Where the /NL switch is used with no argument, the errors and symbol table are output to the device and/or file specified. Use of the switches /NL and /NL:SYM cause only the errors to be sent to the file and/or device specified.

Each argument used with a listing switch is preceded by a colon.

Use of these switches overrides the enabling or disabling of the equivalent listing option in the source. Default listing controls can be specified by the user within his source code and overridden, where necessary, by switch options at assembly time. For example:

```
#OBJFIL,KB:/NL:BEX:COM/LI:SRC<DF:SRCFIL
```

This command string suppresses the listing of binary extensions and source comments and ignores all listing directives with the arguments BEX, COM, and SRC. (The object file is sent to OBJFIL on the system device, and the listing and symbol table to the keyboard.)

```
#OBJFIL,LP:/LI<DT1:ABC
```

causes MACRO to ignore all .LIST and .NLIST directives without arguments. This command string causes the listing of any source code that would have otherwise been suppressed. (The object file is sent to the system device; the source listing and symbol table are sent to the line printer.)

```
#OBJFIL,SYM/NL<ABC
```

causes MACRO to produce only an object file and a symbol table listing. The assembly listing is completely suppressed by the /NL switch. (The object file and symbol table file are sent to the system device.)

5.1.2 Page Headings

The MACRO Assembler outputs each page in the format shown in Figure 6-3 (teleprinter listing). On the first line of each listing page the assembler prints (from left to right) the following items.

1. Title taken from .TITLE directive
2. Assembler version identification
3. Date

4. Time-of-day

5. Page number

The second line of each listing page contains the subtitle text specified in the last encountered .SBTTL directive.

5.1.3 .TITLE

The .TITLE directive is used to assign a name to the object module. The name is the first symbol following the directive and must be six Radix-5Ø characters or less (any characters beyond the first six are ignored). Non-Radix-5Ø characters are not acceptable. For example,

```
.TITLE PROG TO PERFORM DAILY ACCOUNTING
```

causes the object module of the assembled program to be named PROG (this name is distinguished from the filename of the object module specified in the command string to the assembler). The name of the object module appears in the LINK load map and on the listing.

If there is no .TITLE statement, the default name assigned to the object module is

```
.MAIN.
```

The first tab or space following the .TITLE directive is not considered part of the object module name or header text, although subsequent tabs and spaces are significant.

If there is more than one .TITLE directive, the last .TITLE directive in the program conveys the name of the object module.

5.1.4 .SBTTL

The .SBTTL directive is used to provide the elements for a printed table of contents of the assembly listing. The text following the directive is printed as the second line of each of the following assembly listing pages until the next occurrence of a .SBTTL directive. For example:

```
.SBTTL CONDITIONAL ASSEMBLIES
```

The text

```
CONDITIONAL ASSEMBLIES
```

is printed as the second line of each of the following assembly listing pages.

During pass 1 of the assembly process, MACRO automatically prints a table of contents for the listing containing the line sequence number and text of each .SBTTL directive in the program. Such a table of contents is inhibited by specifying the /NL:TOC switch option to the assembly listing file specification (or a .NLIST TOC directive within the source). For example:

```
#OBJFIL,LISTM/NL:TOC<SRCFIL
```

In this case the table of contents normally generated prior to the assembly listing is inhibited.

An example of the table of contents is shown in Figure 6-4. Note that the first word of the subtitle heading is not limited to six characters since it is not a module name.

5.1.5 .IDENT

The .IDENT directive provides another means of labeling the object module produced as a result of a MACRO assembly. In addition to the name assigned to the object module with the .TITLE directive, a character string (up to six characters, treated like a RAD5Ø string) can be specified between paired delimiters. For example:

```
.IDENT  /VØØ5A/
```

The character string

```
VØØ5A
```

is converted to Radix-5Ø notation and included in the global symbol directory of the object module.

This symbol is included in the load map listings output by LINK.

When more than one .IDENT directive is found in a given program, the last .IDENT found determines the symbol which is passed as part of the object module identification.

```
 5-  1          SECTOR INITIALIZATION
 7-  1          SUBROUTINE CALL DEFINITIONS
12-  1          PARAMETERS
14-  1          ROLL DEFINITIONS
16-  1          PROGRAM INITIALIZATION
26-  1          ASSEMBLER PROPER
36-  1          STATEMENT PROCESSOR
4Ø-  1          ASSIGNMENT PROCESSOR
41-  1          OP CODE PROCESSOR
48-  1          EXPRESSION TO CODE-ROLL CONVERSIONS
5Ø-  1          CODE ROLL STORAGE
51-  1          DIRECTIVES
59-  1          DATA-GENERATING DIRECTIVES
68-  1          CONDITIONALS
72-  1          LISTING CONTROL
74-  1          ENABL/DSABL FUNCTIONS
75-  1          CROSS REFERENCE HANDLERS
78-  1          LISTING STUFF
79-  1          KEYBOARD HANDLERS
8Ø-  1          OBJECT CODE HANDLERS
88-  1          LISTING OUTPUT
92-  1          I/O BUFFERS
93-  1          EXPRESSION EVALUATOR
99-  1          TERM EVALUATOR
1Ø3-  1         SYMBOL/CHARACTER HANDLERS
1Ø9-  1         ROLL HANDLERS
114-  1         REGISTER STORAGE
116-  1         MACRO HANDLERS
135-  1         FIN
```

Table of contents text is taken from the text of each .SBTTL directive. The associated numbers are the page and line sequence numbers of the .SBTTL directives.

Figure 6-4
Assembly Listing Table of Contents

5.1.6 Page Ejection

There are three ways of obtaining a page eject in a MACRO assembly listing.

1. After a line count of 58 lines, MACRO automatically performs a page eject to skip over page perforations on line printer paper and to formulate terminal output into pages.

2. More commonly, the .PAGE directive is used within the source code to perform a page eject at that point. The format of this directive appears here.

   ```
   .PAGE
   ```

   This directive takes no arguments and causes a skip to the top of the next page.

   Used within a macro definition, the .PAGE is ignored, but the page eject is performed at each invocation of that macro.

3. The insertion of form feed characters (FF) cause page ejection.

5.2 FUNCTIONS: .ENABL AND .DSABL DIRECTIVES

Several functions are provided by MACRO through the .ENABL and .DSABL directives. These directives use 3-character symbolic arguments to designate the desired function, and are of the forms

```
.ENABL  arg
.DSABL  arg
```

where arg is one of the legal symbolic arguments as described in Table 6-9.

Table 6-9
Functions: Symbolic Arguments

Argument	Default	Function
ABS	disable	Produces absolute binary output; i.e., input to the paper tape software system absolute loader.
AMA	disable	Causes the assembly of all relative addresses (address mode 67) as absolute addresses (address mode 37). This switch is useful during the debugging phase of program development.
CDR	disable	Causes source columns 73 and greater to be treated as comment. This accommodates sequence numbers in card columns 72-8Ø.
FPT	disable	Causes floating point truncation, rather than rounding, as is otherwise performed. .DSABL FTP returns to floating point rounding mode.
LC	disable	Causes the assembler to accept lower case ASCII input instead of converting it to upper case.
LSB	disable	Causes a local symbol block to be started. See Figure 6-5.
PNC	enable	Causes binary output to be produced on the source listing.
REG	enable	Causes the default register names to be defined. The following code is implied as being present. RØ=%Ø Rl=%1 R2=%2 R3=%3 R4=%4 R5=%5 SP=%6 PC=%7
GBL	enable	Causes the assembler to attempt to resolve undefined or global references at the end of pass 1.

```
MACRO   MACRO V06-03 21-FEB-74 03:10 PAGE 15
STATEMENT PROCESSOR

1 000516          LABEL:                              ;LABEL PROCESSOR
2                         .ENABL  LSB
3 000516 026767           CMP     SYMBOL,R50DOT       ;PERIOD?
         000000'
         000004'
4 000524 001470           BEQ     4$                  ;  YES, ERROR
5                         .IF NDF XEDLSB
6 000526                  CALL    LSBSET              ;FLAG START OF NEW LOCAL SYMBOL
7                         .ENDC
8 000532                  SSRCH                       ;NO, SEARCH THE SYMBOL TABLE
9 000536                  CRFDEF
10 00542          LABELF: SETXPR                      ;SET EXPRESSION REGISTERS
11 00546                  GETNB                       ;BYPASS COLON
12 00552 005046           CLR     -(SP)               ;ASSUME NO GLOBAL DEFINITION
13 00554 020527           CMP     R5,#CH.COL          ;SECOND COLON?
         000072
14 00560 001004           BNE     10$                 ;IF NE NO
15 00562 012716           MOV     #GLBFLG,(SP)        ;SET GLOBAL DEFINITION BIT
         000100
16 00566                  GETNB                       ;BYPASS SECOND COLON
17 00572 032713 10$:      BIT     #DEFFLG,(R3)        ;ALREADY DEFINED?
         000010
18 00576 001020           BNE     1$                  ;  YES
19 00600 016700           MOV     CLCFGS,R0           ;NO, GET CURRENT LOCATION CHARAC
         000024'
20 00604 042700           BIC     #377-<RELFLG>,R0          ;CLEAR ALL BUT RELOCATIO
         000337
21 00610 052700           BIS     #DEFFLG!LBLFLG,R0         ;FLAG AS LABEL
         000012
22 00614 051600           BIS     (SP),R0             ;MERGE POSSIBLE GLOBAL DEFINITIO
23 00616 032713           BIT     #DFGFLG,(R3)        ;DEFAULT GLOBAL FROM REF?
         000020
24 00622 001402           BEQ     11$                 ;IF EQ NO
25 00624 042713           BIC     #DFGFLG!GLBFLG,(R3);CLEAR DEFAULT FLAGS
         000120
26 00630 050013 11$:      BIS     R0,(R3)             ;SET MODE BITS
27 00632 016714           MOV     CLCLOC,(R4)         ;  AND CURRENT LOCATION
```

Figure 6-5
Example of .ENABL and .DSABL Directives

```
          000026'
28 00636  000416          BR      3$                ;INSERT
29 00640  032713  1$:     BIT     #LBLFLG,(R3)      ;DEFINED, AS LABEL?
          000002
30 00644  001406          BEQ     2$                ;  NO, INVALID
31 00646  026714          CMP     CLCLOC,(R4)       ;HAS ANYBODY MOVED?
          000026'
32 00652  001003          BNE     2$                ;  YES
33 00654  126712          CMPB    CLCSEC,(R2)       ;SAME SECTOR?
          000025'
34 00660  001405          BEQ     3$                ;  YES, OK
35 00662          2$:     ERROR   P                 ;NO, FLAG ERROR
36 00670  052713          BIS     #MDFFLG,(R3)      ;FLAG AS MULTIPLY DEFINED
          000004
37 00674          3$:     INSERT                    ;INSERT/UPDATE
38 00700                  SETPF0                    ;BE SURE TO PRINT LOCATION FIELD
39 00704  000404          BR      5$
40 00706          4$:     ERROR   Q
41 00714  000401          BR      6$                ;
42 00716  005726  5$:     TST     (SP)+             ;CLEAN STACK

MACRO   MACRO V06-03 21-FEB-74 03:10 PAGE 15-1
STATEMENT PROCESSOR

43 00720          6$:     SETNB                     ;BYPASS ANY BLANKS
44 00724  016767          MOV     CHRPNT,LBLEND     ;MARK END OF LABEL
          000032'
          000000G
45 00732  000167          JMP     STMNT             ;TRY FOR MORE
          177216
46
47                        .DSABL  LSB
```

Figure 6-5 (Cont.)
Example of .ENABL and .DSABL Directives

A misspelled argument causes the directive containing it to be flagged as an error. No further action is taken. These functions can also be controlled through switches specified in the command string to the MACRO Assembler. The switches are

```
/EN:arg
/DS:arg
```

where arg is any of the arguments that can be used with the .ENABL and .DSABL directives.

Use of these switches overrides the enabling or disabling of all occurrences of that argument in the program. They are used in the same manner as /LI, /NL, but in general apply mainly to source files.

5.3 DATA STORAGE DIRECTIVES

The MACRO Assembler generates a wide range of data and data types. These facilities are explained in the following sections.

5.3.1 .BYTE

The .BYTE directive is used to generate one or more successive bytes of data.

Format:

```
.BYTE[exp1][,exp2,...]
```

A legal expression must have an absolute value (or contain a reference to an external symbol) and must result in eight bits or less of data. The 16-bit value of the expression must have a high-order byte (which is truncated) that is either all zeros or all ones. Each operand expression is stored in a byte of the object program. Multiple operands are separated by commas and stored in successive bytes. For example:

```
SAM=5
.=41Ø
        .BYTE ↑D48,SAM      ;Ø6Ø (OCTAL EQUIVALENT OF 48
                            ;DECIMAL) IS STORED IN LOCATION
                            ;41Ø, ØØ5, IS STORED IN
                            ;LOCATION 411.
```

If the high-order byte of the expression equates to a value other than Ø or -1, it is truncated to the low-order eight bits and flagged with a T error code. If the expression is relocatable, an A-type warning flag is given.

At link time it is likely that relocation will result in an expression of more than eight bits, in which case, LINK prints a truncation error message. For example:

```
        .BYTE   23          ;STORES OCTAL 23 IN NEXT BYTE.
A:
        .BYTE   A           ;RELOCATABLE VALUE CAUSES AN "A"
                            ;ERROR FLAG.

        .GLOBL  X
X=3
        .BYTE   X           ;STORES 3 IN NEXT BYTE.
```

If an operand following the .BYTE directive is null, it is interpreted as a zero. For example:

```
.=42Ø
        .BYTE,,             ;ZEROS ARE STORED IN BYTES 42Ø, 421,
                            ;AND 422.
```

5.3.2 .WORD

The .WORD directive is used to generate one or more successive words of data.

Format:

```
.WORD[exp1][,exp2,...]
```

A legal expression must result in 16 bits or less of data. Each operand expression is stored in a word of the object program. Multiple operands are separated by commas and stored in successive words. For example:

```
SAL=Ø
.=5ØØ
        .WORD   177535,.+4,SAL      ;STORES 177535, 5Ø6 AND Ø IN
                                    ;WORDS 5ØØ, 5Ø2 AND 5Ø4.
```

If an expression equates to a value of more than 16 bits, it is truncated and flagged with a T error code.

If an operand following the .WORD directive is null, it is interpreted as zero. For example:

```
.=5ØØ
        .WORD   ,5,                 ;STORES Ø, 5, AND Ø IN LOCATIONS
                                    ;5ØØ, 5Ø2, and 5Ø4
```

A blank operator field (any operator not recognized as a macro call, op-code, directive or semicolon) is interpreted as an implicit .WORD directive. Use of this convention is discouraged because it may not be the default case in future PDP-11 assemblers. The first term of the first expression in the operand field must not be an instruction mnemonic or assembler directive unless preceded by a + or - operator. For example:

```
.=44Ø                   ;THE OP-CODE FOR MOV, WHICH
                        ;IS Ø1ØØØØ, IS STORED ON
LABEL:  +MOV,LABEL      ;LOCATION 44Ø.  44Ø IS
                        ;STORED IN LOCATION 442.
```

Note that the default .WORD directive occurs whenever there is a leading arithmetic or logical operator; or whenever a leading symbol is encountered that is not recognized as a macro call, an instruction mnemonic, or an assembler directive. Therefore, if an instruction mnemonic, macro call, or assembler directive is misspelled, the .WORD directive is assumed and errors will result. Assume that MOV is spelled incorrectly as MOR.

```
MOR     A,B
```

Two error codes result: Q occurs because an expression operator is missing between MOR and A, and U occurs if MOR is undefined. The U error occurs only if GBL is disabled and MOR is undefined, else MOR is classed as a global. Two words are then generated: one for MOR A and one for B.

5.3.3 ASCII Conversion of One or Two Characters

The ' and " characters are used to generate text characters within the source text. A single apostrophe followed by a character results in a word in which the 7-bit ASCII representation of the character is placed in the low-order byte and zero is placed in the high-order byte. For example,

```
MOV     #'A,RØ
```

results in the following 16 bits being moved into RØ.

ØØØØ ØØØØ	Ø1ØØ ØØØ1

ASCII value of A

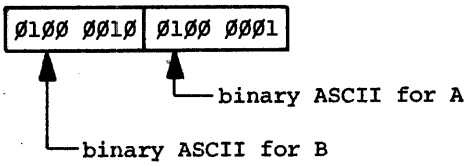
```
STMNT:
        GETSYM
        BEQ     4$
        CMPB    @CHRPNT,#':     ;COLON DELIMITS LABEL FIELD.

        BEQ     LABEL
        CMPB    @CHRPNT,#'=     ;EQUAL DELIMITS
        BEQ     ASGMT           ;ASSIGNMENT PARAMETER.
```

A double quote allows for the 7-bit ASCII representation of two characters to be placed in the low and high order bytes of a word. For example,

```
.WORD  "AB
```

results in the creation of the following binary word constant.



5.3.4 .ASCII

The .ASCII directive translates character strings into their 7-bit ASCII equivalents for use in the source program.

Format:

```
.ASCII /character string/
```

where

character string	is a string of any acceptable printable ASCII characters. The string may not include null, rubout, return, line feed, vertical tab, or form feed characters. Nonprinting characters can be expressed in digits of the current radix and delimited by angle brackets. Any legal, defined expression is allowed between angle brackets.
/ /	these are delimiting characters and may be any printing characters other than ; < = or any character within the string.

As an example:

```
A:      .ASCII /HELLO/              ;STORES ASCII REPRESENTATION OF
                                    ;THE LETTERS H E L L O IN 5
                                    ;CONSECUTIVE BYTES.

        .ASCII /ABC/<15><12>/DEF/   ;STORES A B C 15(8) 12(8) D E F IN 8
                                    ;CONSECUTIVE BYTES.

        .ASCII /<AB>/               ;STORES < A B > IN 4 CONSECUTIVE
                                    ;BYTES.
```

The ; and = characters are not illegal delimiting characters, but are pre-empted by their significance as a comment indicator and assignment operator, respectively. For other than the first group, semicolons are treated as beginning a comment field. For example:

```
.ASCII   ;ABC;/DEF/          ;STORES A B C D E F
                             ;NOT RECOMMENDED PRACTICE

.ASCII   /ABC/;DEF;          ;STORES A B C.  DEF TREATED
                             ;AS A COMMENT

.ASCII   /ABC/=DEF=          ;SAME AS CASE 1

.ASCII   =DEF=               ;THE ASSIGNMENT
                             ;.ASCII=DEF
                             ;IS PERFORMED AND A Q ERROR GENERATED
                             ;UPON ENCOUNTERING
                             ;THE SECOND =.
```

5.3.5 .ASCIZ

The .ASCIZ directive is equivalent to the .ASCII directive with a zero byte automatically inserted as the final character of the string. For example:

When a list or text string has been created with a .ASCIZ directive, a search for the null character can determine the end of the list. For example:

```
1
2
3
4
5
6
7
8                       ;       CALLED BY JSR PC,EX1
9
10
11 00000 012701 EX1:    MOV     #HELLO,R1
         000016'
12 00004 012702         MOV     #LINBUF,R2
         000030'
13 00010 112122 X:      MOVB    (R1)+,(R2)+     ;MOVE DATA
14 00012 001376         BNE     X
15 00014 000207         RTS     PC
16       000015 CR=15
17       000013 LF=13
18 00016    015 HELLO:  .ASCIZ  <CR><LF>/HELLO/<CR><LF>
   00017    013
   00020    110
   00021    105
   00022    114
   00023    114
   00024    117
   00025    015
   00026    013
   00027    000
19 00030        LINBUF: .BLKW   6
```

5.3.6 .RAD5Ø

The .RAD5Ø directive allows the user the capability to handle symbols in Radix-5Ø coded form (this form is sometimes referred to as MOD4Ø and is used in PDP-11 system programs). Radix-5Ø form allows three characters to be packed into sixteen bits; therefore, any 6-character symbol can be held in two words.

Format:

```
.RAD5Ø    /string/
```

where

/ /	delimiters can be any printing characters other than the = < and ; characters.
string	is a list of the characters to be converted (three characters per word) and which may consist of the characters A through Z, Ø through 9, $. and space. If there are fewer than three characters (or if the last set is fewer than three characters) they are considered to be left justified and trailing spaces are assumed. Illegal nonprinting characters are replaced with a ? character and cause an I error flag to be set. Illegal printing characters set the Q error flag.

The trailing delimiter may be a semicolon or matching delimiter. For example:

```
.RAD5Ø    /ABC/       ;PACK ABC INTO ONE WORD.
.RAD5Ø    /AB/        ;PACK AB (SPACE) INTO ONE WORD.
.RAD5Ø    /ABCD/      ;PACK ABC INTO FIRST WORD AND
                      ;D SPACE SPACE INTO SECOND WORD.
```

Each character is translated into its Radix-5Ø equivalent as indicated below.

Character	Radix-5Ø Equivalent (octal)
(space)	Ø
A-Z	1-32
$	33
.	34
Ø-9	36-47

The character code for 35 is currently undefined.

The Radix-5Ø equivalents for characters 1 through 3 (C1,C2,C3) are combined as follows.

$$\text{Radix 5Ø value} = ((C*50_8)+C2)*50_8+C3$$

For example:

Radix-5Ø value of ABC is $((1*50_8)+2)*50_8+3$ or 3223

See Appendix B for a table of Radix-50 equivalents.

Use of angle brackets is encouraged in the .ASCII, .ASCIZ, and .RAD50 statements whenever leaving the text string to insert special codes. For example:

```
1
2
3
4
5
6
7
8
9  000044 003223            .RAD50  /ABC/              ;STORES 3223
10 000046 003223            .RAD50  /AB/<3>            ;EQUIVELENT TO /ABC/
11         000001 V1=1
12         000002 V2=2
13         000003 V3=3
14 00050 003223             .RAD50  <V1><V2><V3>
```

5.4 RADIX CONTROL

5.4.1 .RADIX

Numbers used in a MACRO source program are initially considered to be octal numbers. However, the programmer has the option of declaring the radices 2, 4, 8, 10. This is done via the .RADIX directive,

```
.RADIX [n]
```

where n is one of the acceptable radices.

The argument to the .RADIX directive is always interpreted in decimal radix. Following any radix directive, that radix is the assumed base for any number specified until the following .RADIX directive.

The default radix at the start of each program, and the argument assumed if none is specified, is 8 (i.e., octal). For example:

```
.RADIX 10               ;BEGINS SECTION OF CODE WITH
                        ;DECIMAL
                        ;RADIX
   .
   .
   .
.RADIX                  ;REVERTS TO OCTAL RADIX
```

In general it is recommended that macro definitions not contain or rely on radix settings from the .RADIX directive. The temporary radix control characters should be used within a macro definition and are described in the following section. A

given radix is valid throughout a program until changed. Where a possible conflict exists within a macro definition or in possible future uses of that code module, it is suggested that the user specify values using the temporary radix controls.

5.4.2 Temporary Radix Control: ↑D, ↑O, and ↑B

Once the user has specified a radix for a section of code, or has determined to use the default octal radix, he may discover a number of cases where an alternate radix is more convenient (particularly within macro definitions). For example, the creation of a mask word might best be done in the binary radix.

MACRO has three unary operators to provide a single interpretation in a given radix within another radix.

```
↑Dx  (x is treated as being in decimal radix)
↑Ox  (x is treated as being in octal radix)
↑Bx  (x is treated as being in binary radix)
```

For example:

```
↑D123
↑O 47
↑B ØØØØ11Ø1
↑O<A+3.>
```

Notice that while the up arrow and radix specification characters may not be separated, the radix operator can be physically separated from the number by spaces or tabs for formatting purposes. Where a term or expression is to be interpreted in another radix, it should be enclosed in angle brackets.

These numeric quantities may be used any place where a numeric value is legal.

MACRO provides a feature (maintained for compatibility with PAL-11) that allows temporary radix change from octal to decimal by specifying a decimal radix number with a decimal point. For example,

```
 1ØØ.
1376.
 128.
```

are all decimal numbers.

5.5 LOCATION COUNTER CONTROL

Four directives control movement of the location counter. .EVEN and .ODD move the counter a maximum of one byte. .BLKB and .BLKW allow the user to specify blocks of a given number of bytes or words to be skipped in the assembly.

5.5.1 .EVEN

The .EVEN directive ensures that the assembly location counter contains an even memory address by adding one if the current address is odd. If the assembly location counter is even, no action is taken. Any operands following an .EVEN directive are ignored.

5.5.2 .ODD

The .ODD directive ensures that the assembly location counter is odd by adding one if it is even. For example:

```
1
2
3
4
5
6
7
8 000052     001 ODEVE:  .BYTE   1,2,3
  000053     002
  000054     003
9                        .EVEN                    ;ADJUST TO EVEN BOUNDRY
10 00056     001         .BYTE   1,2
   00057     002
11                       .ODD                     ;ADJUST TO ODD BOUNDRY
12 00061     001         .BYTE   1,2,3
   00062     002
   00063     003
13                       .EVEN                    ;EVEN BOUNDRY?
14 00064     001         .BYTE   1,2,3
   00065     002
   00066     003
15                       .ODD                     ;ODD BOUNDRY?
16 00067     001         .BYTE   1,2,3
   00070     002
   00071     003
17                       .EVEN
```

5.5.3 .BLKB and .BLKW

Blocks of storage can be reserved using the .BLKB and .BLKW directives. .BLKB is used to reserve byte blocks and .BLKW reserves word blocks.

Format:

```
.BLKB   [exp]
.BLKW   [exp]
```

where exp is the number of bytes or words to reserve. If no expression is present, 1 is the assumed default value. Any legal expression that is completely defined at assembly time and produces an absolute number is legal; e.g., external expressions are illegal. Using these directives without arguments is not recommended.

For example:

```
1          000000'          .CSECT  IMPURE
2
3 000000          PASS:     .BLKW
4                                              ;NEXT GROUP MUST STAY TOGETHER
5 000002          SYMBOL:   .BLKW   2          ;SYMBOL ACCUMULATOR
6 000006          MODE:
7 000006          FLAGS:    .BLKB   1          ;FLAG BITS
8 000007          SECTOR:   .BLKB   1          ;SYMBOL/EXPRESSION TYPE
9 000010          VALUE:    .BLKW   1          ;EXPRESSION VALUE
10 000012         RELLVL:   .BLKW   1
11                          .BLKW   2          ;END OF GROUPED DATA
12
13 000020         CLCNAM:   .BLKW   2          ;CURRENT LOCATION COUNTER SYMBOL
14 000024         CLCFGS:   .BLKB   1
15 000025         CLCSEC:   .BLKB   1
16 000026         CLCLOC:   .BLKW   1
17 000030         CLCMAX:   .BLKW   1
```

The .BLKB directive has the same effect as

```
.=.+exp
```

but is easier to interpret in the context of source code.

5.6 NUMERIC CONTROL

Several directives are available to simplify the use of the floating-point hardware on the PDP-11. (Refer to Processor Handbook for floating-point hardware description.)

A floating-point number is represented by a string of decimal digits. The string may contain an optional decimal point and an optional exponent indicator (the letter E and a signed decimal exponent). The list below contains seven valid representations of the same floating-point number:

```
3
3.
3.0
3.0E0
3E0
.3E1
300E-2
```

The list could be extended indefinitely (e.g., 3000E-3, .03E2, etc.). A leading plus sign is ignored (e.g., +3.0 is considered to be 3.0). A leading minus sign complements the sign bit. No other operators are allowed (e.g., 3.0+N is illegal).

Floating-point number representations are valid only in the contexts described in the remainder of this section.

Floating-point numbers are normally rounded. That is, when a floating-point number exceeds the limits of the field in which it is to be stored, the high-order excess bit is added to the low-order retained bit. For example, if the number is to be stored in a 2-word field, but more than 32 bits are needed for its value, the highest bit carried out of the field is added to the least significant position. The .ENABL FPT directive is used to enable floating-point truncation, and .DSABL FPT is used to return to floating-point rounding (see Section 6-5.2).

5.6.1 .FLT2 and .FLT4

Like the .WORD directive, the two floating-point storage directives cause their arguments to be stored in line with the source program.

Format:

```
.FLT2   arg1,arg2,...
.FLT4   arg1,arg2,...
```

where arg1,arg2,... represent one or more floating point numbers separated by commas.

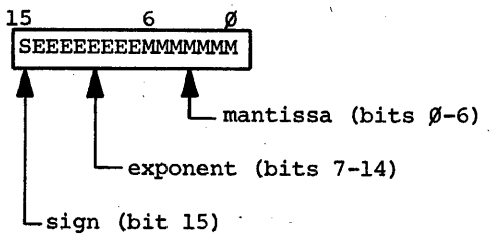
.FLT2 causes two words of storage to be generated for each argument, while .FLT4 generates four words of storage.

5.6.2 Temporary Numeric Control: ↑F and ↑C

Like the temporary radix control operators, operators are available to specify either a 1-word floating-point number (↑F) or the 1's complement of a 1-word number (↑C). The ↑F operator can only be used within an instruction operand expression. ↑C can be used in any expression. For example,

```
FL3.7:  MOV     #↑F3.7,RØ
```

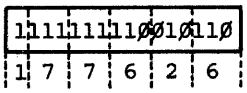
creates a 1-word floating-point number at location FL3.7+2, containing the value 3.7 formatted as follows.



This 1-word floating-point number is similar to the first word of a 2- or 4-word floating-point number format shown in the PDP-11 Processor Handbook. The statement

```
CMP151: .WORD    ↑C151
```

stores the 1's complement of 151 in the current radix (assume current radix is octal) as follows (177626 shown in binary)



Since these control operators are unary operators, their arguments may be terms, and the operators may be expressed recursively. For example:

```
↑F<1.2E3>
↑C<D25>      or   ↑C31       or   177746
```

The term created by the unary operator and its argument is then a term that can be used by itself or in an expression. For example,

```
↑C2+6 is equivalent to 177775
```

while

```
↑<C2>+6 is equivalent to   177775+6  or  000003.
```

For this reason, the use of angle brackets is advised. Expressions used as terms or arguments of a unary operator must be explicitly grouped.

An example of the importance of ordering with respect to unary operators is shown below.

```
↑F1.0   = 02040000
↑F-1.0  = 12040000

-↑F1.0   = 15740000
-↑F-1.0  = 05740000
```

The argument of the ↑F operator must not be an expression and must be of the same format as arguments to the .FLT2 and .FLT4 directives (see Section 6-5.6.1).

5.7 TERMINATING DIRECTIVES

5.7.1 .END

The .END directive indicates the physical end of the source program.

Format:

```
.END    [exp]
```

where exp is an optional argument that indicates the program entry point; i.e., the transfer address.

When the load module is loaded, program execution begins at the transfer address indicated by the .END exp directive.

5.7.2 .EOT

Under the DOS/BATCH Monitor, the .EOT directive is ignored.

5.8 PROGRAM BOUNDARIES DIRECTIVE: .LIMIT

It is often important to know the boundaries of the load module's relocatable code. The .LIMIT directive reserves two words into which LINK puts the low and high addresses of the relocated code. The low address (inserted into the first word) is the address of the first byte of code. The high address is the address of the first free byte following the relocated code.

Example:

```
1
2
3
4
5
6
7  000072 000000'          .LIMIT              ;LIMIT AND LIMIT+2
   000074 000000
8                                              ;WILL BE THE PROGRAM LIMITS
```

5.9 PROGRAM SECTION DIRECTIVES

The assembler provides for 255 program sections: one absolute section, one blank relocatable section, and 253 named relocatable sections. The .PSECT directive enables the user to create his program (object module) in sections, and share code and data.

5.9.1 .PSECT Directive

Program sections are defined by the .PSECT directive.

Format:

```
.PSECT  [NAME][,RO/RW][,I/D][,GBL/LCL][,ABS/REL][,CON/OVR][,HGH/LOW]
```

Any program section without a .PSECT directive is given the name .MAIN., and is assigned all the default attributes. Table 6-10 summarizes the program section attributes.

Table 6-10
.PSECT Directive Parameters

Parameter	Default	Meaning to Linker
NAME	blank	Program section name, in Radix-5Ø format, specified as one to six characters. If omitted, a comma must appear in the first parameter's position.
RO/RW	RW	Defines the type of access to the program section permitted; read only or read/write.
I/D	I	Allows LINK to differentiate global symbols that are entry points (I) from global symbols that are data values (D).
GBL/LCL	LCL	Defines the scope of a program section. A global program section's scope crosses segment (overlay) boundaries; a local program section's scope is within a single segment. In single-segment programs, the GBL/LCL parameter is ignored.
ABS/REL	REL	When ABS is specified, the program section is absolute. No relocation is performed by the Linker for references within that section. When REL is specified, a relocation bias is calculated by LINK, and added to all references in the section.
CON/OVR	CON	CON causes LINK to collect all allocation references to the program section from different modules and concatenate them to form the total allocation for the program section. OVR indicates that all allocation references to the program section overlay one another. Thus, the total allocation of the program section is determined by the largest request made by a module that references it.
HGH/LOW	LOW	Program section memory type. HGH = high-speed LOW = core

NOTE

The HGH/LOW attribute is currently ignored by LINK.

The first parameter must always be NAME. If it is omitted, a comma must be used in its place.

Example:

```
.PSECT ,RO
```

This example shows a .PSECT with a blank name and the read only access parameter. Defaults are used for the remaining parameters.

Once the attributes of a named .PSECT are declared in a module, the MACRO Assembler assumes that this .PSECT's attributes hold for all subsequent declarations of the named .PSECT in the same module. Thus, the attributes may be declared once, and later .PSECT's with the same name will have the same attributes, when specified within the same module.

For each program section specified or implied, the assembler maintains the following information.

1. Section name
2. Contents of the program counter
3. Maximum program counter value encountered
4. Section attributes (the six .PSECT attributes)

5.9.1.1 Creating Program Sections

The attributes of a given program section are defined by explicit and default parameters upon its first reference. Thereafter, references to the section can either respecify the same section attributes, or the section name only. You may not assign different attributes on a later call to the section. For example, a section can be specified as

```
.PSECT    ALPHA,ABS,OVR
```

which will give it the attributes ALPHA,RW,I,LCL,ABS,OVR,LOW. The same program can be later referenced as

```
.PSECT    ALPHA
```

and the same attributes will still be in effect.

By maintaining separate location counters for each section, the assembler allows the user to write statements which are not physically contiguous but are loaded contiguously, as shown in the following example.

```
1
2
3
4
5
6
7
8
9
10                      ;       CALLED BY       JSR     PC,CLRR
11
12              000000'         .PSECT  CLEAR,REL       ;SECT. CLEAR RELOCATABLE
13  00000 000000 A:             .WORD   0               ;POINTER VARIABLES
14  00002 000000 B:             .WORD   0
15  00004 000000 C:             .WORD   0
16  00006 000000 D:             .WORD   0
17
18  00010 005067 CLRR:          CLR     A               ;SET TO NULL
            177764
19  00014 005067                CLR     B
            177762
20  00020 005067                CLR     C
            177760
21
22              000000'         .PSECT  VECT,ABS        ;SECT.VECT ABSOLUTE
23              000004'         .=.+4
24  00004 000000G               .WORD   TRAPP
25  00006 000360                .WORD   360             ;PRIORITY 7
26
27              000024'         .PSECT  CLEAR           ;SECT. CLEAR
28  00024 005067                CLR     D
            177756
29  00030 000207                RTS     PC              ;RETURN
```

The first appearance of a .PSECT directive with a given name assumes the location counter is at relocatable or absolute zero. The scope of each directive extends until a directive beginning a different section is given. Further occurrences of a section name in a subsequent .PSECT statement resume assembling where the section previously ended.

All labels in an absolute section are absolute; all labels in a relocatable section are relocatable. The location counter symbol . is relocatable or absolute when referenced in a relocatable or absolute section, respectively. An undefined symbol is a global reference. It essentially has no attributes except global reference.

Any labels appearing on a .PSECT (or .ASECT or .CSECT) statement are assigned the value of the location counter before the .PSECT (or other) directive takes effect. Thus, if the first statement of a program is

```
A:      .PSECT  ALT,REL
```

then A is assigned to relocatable zero and is associated with the relocatable section ALT.

Since it is not known at assembly time where the program sections are to be loaded, all references between sections in a single assembly are translated by the assembler to references relative to the base of that section. The assembler provides LINK with the necessary information to resolve the linkage. This information is not necessary when making a reference to an absolute section. The assembler can determine all load addresses of an absolute section.

In the following example, references to X1 and Y are translated into references relative to the base of the relocatable section P2.

```
1
2
3
4
5
6          000000'          .PSECT  P1,ABS  ;SECT. P1 - ABSOLUTE
7 000000 005067 AAA:        CLR     X1      ;CLEAR X1 IN RELOCATABLE SECTION
           000004'
8 000004 000167             JMP     Y       ;GOTO RELOCATABLE SECTION
           000000'
9
10         000000'          .PSECT  P2,REL  ;SECT. P2 - RELOCATABLE
11 00000 000167 Y:          JMP     AAA     ;GOTO ABSOLUTE SECTION
           000000'
12 00004 000000 X1:         .WORD   0
```

5.9.1.2 Code or Data Sharing

Named relocatable program sections with the attribute OVR can be used to redefine the same sections of core. Sections of the same name with the attribute OVR from different assemblies are all loaded at the same location by LINK. All other program sections (those with the attribute CON) are concatenated.

Note that there is no conflict between internal symbolic names and program section names. It is legal to use the same symbolic name for both purposes. Program section names should not duplicate .GLOBL names.

5.9.2 .ASECT and .CSECT Directives

DOS/BATCH assembly language programs use the .PSECT directive exclusively, since it affords all the capabilities of the .ASECT and .CSECT directives defined for other PDP-11 assemblers. For the sake of compatibility with non-DOS/BATCH MACRO programs, the MACRO Assembler will accept .ASECT and .CSECT directives, but assembles them as if they were .PSECT's with the default attributes listed in Table 6-11.

Table 6-11
Non-DOS/BATCH Program Section Defaults

Attribute	Default Value		
	.ASECT	.CSECT (named)	.CSECT
Name	ABS	name	blank
Access	RW	RW	RW
Type	I	I	I
Scope	GBL	GBL	LCL
Relocation	ABS	REL	REL
Allocation	OVR	OVR	CON
Memory	LOW	LOW	LOW

The allowable syntactical forms of .ASECT and .CSECT follow here.

```
.ASECT
.CSECT [symbol]
```

Note that, due to default attribute selection applied to .CSECT's by MACRO,

```
.CSECT  JIM
```

is identical to

```
.PSECT  JIM,GBL,OVR,RW,I,REL,LOW.
```

5.10 SYMBOL CONTROL: .GLOBL

The assembler produces a relocatable object module and a listing file containing the assembly listing and its associated symbol table. LINK joins separately assembled object modules into a single load module. Object modules are relocated as a function of the specified base of the load module. The object modules (where there are more than one) are linked via global symbols such that a global

symbol in one module (either defined by direct assignment or as a label) can be referenced from another module.

A global symbol may be specified in a .GLOBL directive.

In addition, symbols referenced but not defined within a module are assumed to be global references. The .GLOBL directive is provided to reference (and provide linkage to) symbols not otherwise referenced within a module. For example, one might include a .GLOBL directive to cause linkage to a library. When defining a global definition, the .GLOBL A,B,C directive is equivalent to the following.

```
A==value (or A::value)
B==value (or B::value)
C==value (or C::value)
```

The form of the .GLOBL directive is

```
.GLOBL    sym1,sym2,...
```

where sym1,sym2,... are legal symbolic names, separated by commas or spaces where more than one symbol is specified.

Symbols appearing in a .GLOBL directive are either defined within the current program, or are external symbols defined in another program. This other program is linked with the current program by LINK prior to execution in order to resolve all references to external symbols.

A .GLOBL directive line may contain a label in the label field and comments in the comment field.

At the end of assembly pass 1, MACRO has determined whether a given global symbol is defined within the program or is an external symbol. All internal symbols to a given program must be defined by the end of pass 1, or they will be assumed to be global references (see .ENABL, .DSABL or globals in Section 6-5.2).

```
1
2
3
4
5
6
7                       ;       ROUTINE WITH TWO ENTRIES
8                       ;       DEPENDING ON NUMBER OF FOLLOWING ARGUMENTS
9                       ;       CALLED BY  JSR R5,EN3 FOR THREE ARGS.
10                      ;       CALLED BY  JSR R5,EN2 FOR TWO ARGS.
11
12 000006 013546 EN3:   MOV     @(R5)+,-(SP)    ;GET FIRST PARAM
13 000010 004767        JSR     PC,CONV         ;CONVERT IT
           000000'
14 000014 012075        MOV     (SP)+,@(R5)     ;PASS IT BACK
           000000
15 000020 012535 EN2:   MOV     (R5)+,@(R5)+    ;SEND PARAM
16 000022 000205        RTS     R5              ;RETURN
```

References to external symbols can appear in the operand field of an instruction or an assembler directive in the form of a direct reference,

```
.GLOBL  EXT
CLR     EXT
.WORD   EXT
CLR     @EXT
```

or a direct reference plus or minus a constant.

```
.GLOBL  EXT
A=6
CLR     EXT+A
.WORD   EXT-2
CLR     @EXT+A
```

An external symbol cannot be used in the evaluation of a direct assignment expression. Exception: a global symbol defined within the program can be used in the evaluation of a direct assignment statement.

5.11 CONDITIONAL ASSEMBLY DIRECTIVES

Conditional assembly directives provide the programmer with the capability to conditionally include or ignore blocks of source code in the assembly process. This technique is used to allow several variations of a program to be generated from the same source program.

The general form of a conditional block follows.

```
.IF     cond,argument(s)    ;START CONDITIONAL BLOCK
            .               ;RANGE OF CONDITIONAL
            .
            .               ;BLOCK
.ENDC                       ;END CONDITIONAL BLOCK
```

where

cond	is a condition which must be met if the block is to be included in the assembly. These conditions are defined in Table 6-12.
argument(s)	are a function of the condition to be tested.
range	is the body of code that is included in the assembly (or ignored) depending upon whether the condition is met. Conditional assembly blocks must end with the .ENDC directive, and are limited to a nesting depth of 16 levels.

Table 6-12
Conditional Assembly Directives

Conditions			
POSITIVE	COMPLEMENT	ARGUMENTS	ASSEMBLE BLOCK IF
EQ	NE	expression	expression=Ø (or Ø)
GT	LE	expression	expression>: (or <Ø)
LT	GE	expression	expression<Ø (or >Ø)
DF	NDF	symbolic argument	symbol is defined (or undefined)
B	NB	macro-type argument*	argument is blank (or nonblank)
IDN	DIF	two macro-type arguments separated by a comma	arguments identical (or different)
Z	NZ	expression	same as EQ/NE
G	L	expression	same as GT/LE

*A macro-type argument is enclosed in angle brackets or within an up-arrow construction (as described in Section 6-6.3.1). For example:

```
<A,B,C>
↑/124/
```

```
1
2
3
4
5
6
7         000000 ALPHA=0
8                        .IF      DF,ALPHA
9         000001         BETA=1
10                       .ENDC
11                       .IF      EQ,ALPHA
12        000377         BETA=377
13                       .ENDC
```

Within the conditions DF and NDF the following two operators are allowed to group symbolic arguments.

&	logical AND operator
!	logical inclusive OR operator

```
1
2
3
4
5
6
7          000000 SYM3=0
8                                 .IF       NDF,SYM1!SYM2!SYM3
9          000010                 ALPHA=10
10                                .ENDC
11                                .IF       NDF,SYM1!SYM2
12         000015                 ALPHA=15
13                                .ENDC
```

5.11.1 Subconditionals

Subconditionals may be placed within conditional blocks to indicate the following.

1. Assembly of an alternate body of code when the condition of the block indicates that the code within the block is not to be assembled.

2. Assembly of a noncontiguous body of code within the conditional block depending upon the result of the conditional test to enter the block.

3. Unconditional assembly of a body of code within a conditional block.

These subconditional directives are defined in Table 6-13.

Table 6-13
Subconditional Directives

Directive	Function
.IFF	The code following this statement up to the next subconditional or end of the conditional block is included in the program, provided the value of the condition tested upon entering the conditional block was false.
.IFT	The code following this statement up to the next subconditional or end of the conditional block is included in the program, provided the value of the condition tested upon entering the conditional block was true.
.IFTF	The code following this statement up to the next subconditional or the end of the conditional block is included in the program, regardless of the value of the condition tested upon entering the conditional block.

The implied argument of a subconditional directive is the value of the condition upon entering the conditional block. Subconditionals are used within outer level conditional blocks. They are ignored within nested, unsatisfied conditional blocks.

```
1
2
3
4
5
6
7
8          000001          SYM1=1
9                          .IF     DF,SYM1
10                         .IFF
11                         MOV     R1,R2       ;ASSEMBLE IF SYM1 UNDEFINED
12                         .IFT
13 00032 010203            MOV     R2,R3       ;ASSEMBLE IF SYM1 DEFINED
14                         .IFTF
15 00034 010304            MOV     R3,R4       ;ASSEMBLE ALWAYS
16                         .ENDC
17
18                         .IF     NE,SYM1
19                         .IFF
20                         MOV     R4,R5       ;ASSEMBLED IF SYM1 = 0
21                         .IFT
22 00036 010501            MOV     R5,R1       ;ASSEMBLED IF SYM1 # 0
23                         .ENDC
```

5.11.2 Immediate Conditionals

An immediate conditional directive is a facility for writing a conditional block in one line. In this form, no .ENDC statement is required and the condition is completely expressed on the line containing the conditional directive. Immediate conditions are of the form

```
.IIF cond, arg, statement
```

where

cond	is one of the legal conditions defined for conditional blocks in Table 6-12.
arg	is the argument associated with the conditional specified; that is, either an expression, symbol, or macro-type argument, as described in Table 6-12.
statement	is the statement to be assembled if the condition is met.

For example:

```
.IIF    DF FOO BEQ ALPHA
```

This statement generates the code

```
BEQ     ALPHA
```

if the symbol FOO is defined.

A label must not be placed in the label field of the .IIF statement. Any necessary labels may be placed on the previous line.

```
LABEL:

        .IIF     DF FOO,BEQ ALPHA

        .IIF     DF FOO, LABEL:  BEQ    ALPHA
```

5.11.3 PAL-11R Conditional Assembly Directives

In order to maintain compatibility with programs developed under PAL-11R, the following conditionals (see Table 6-14) remain permissible under MACRO. It is advisable that future programs be developed using the format for MACRO conditional assembly directives.

Table 6-14
PAL-11R Compatible Directives

Directive	Arguments	Assemble Block if
.IFZ or .IFEQ	expression	expression=Ø
.IFNZ or .IFNE	expression	expression not equal Ø
.IFL or .IFLT	expression	expression<Ø
.IFG or .IFGT	expression	expression>Ø
.IFLE	expression	expression is < or =Ø
.IFDF	logical expression	expression is true (defined)
.IFNDF	logical expression	expression is false (undefined)

The rules governing the usage of these directives are the same as those for the MACRO conditional assembly directives previously described.

PART 6
CHAPTER 6
MACRO DIRECTIVES

6.1 MACRO DEFINITION

In assembly language programming, it is often convenient to generate a recurring code sequence by means of a single statement. In order to do this, the desired coding sequence is first defined as a prototype with dummy arguments. This prototype definition is known as a macro. Once a macro has been defined, it is invoked by a single statement that contains its name and, optionally, a list of real arguments that replaces the corresponding dummy arguments in the prototype definition. Calling a macro causes its prototype statements to be generated in line, replacing the single macro call.

6.1.1 .MACRO

The first statement of a macro definition must be a .MACRO directive. The .MACRO directive is of the form

```
.MACRO name[, dummy argument list]
```

where

name	is the name of the macro. This name is any legal symbol. The name chosen may be used as a label elsewhere in the program.
	represents any legal separator (generally a comma or space).
dummy argument list	represents zero, one, or more legal symbols which may appear anywhere in the body of the macro definition, even as a label. These symbols can be used elsewhere in the user program with no conflicts of definition. Where more than one dummy argument is used, they are separated by any legal separator (generally a comma).

A comment may follow the dummy argument list in a statement containing a .MACRO directive. For example:

```
.MACRO ABS,A,B          ;DEFINE MACRO ABS WITH TWO ARGUMENTS
```

A label must not appear on a .MACRO statement. Labels are sometimes used on macro calls, but serve no function when attached to .MACRO statements.

6.1.2 .ENDM

The final statement of every macro definition must be an .ENDM directive of the form

```
.ENDM[name]
```

where

name	is an optional argument, being the name of the macro terminated by the statement.

For example:

```
.ENDM       (terminates the current macro definition)
.ENDM ABS   (terminates the definition of the macro ABS)
```

If specified, the symbolic name in the .ENDM statement must correspond to the one in the matching .MACRO statement. Otherwise the statement is flagged and processing continues. Specification of the macro name in the .ENDM statement permits the assembler to detect missing .ENDM statements or improperly nested macro definitions.

The .ENDM statement may contain a comment field, but must not contain a label.

Example:

```
1
2
3
4
5
6
7                       .MACRO  TYPE,MESS
8                       JSR     R5,TYPE ;OUTPUT MESSAGE
9                       .WORD   MESS
10                      .ENDM
11
12 00040                TYPE    HELLO
   00040 004567         JSR     R5,TYPE ;OUTPUT MESSAGE
         000000G
   00044 000016'        .WORD   HELLO
13
```

6.1.3 .MEXIT

In order to implement alternate exit points from a macro (particularly nested macros), the .MEXIT directive is provided. .MEXIT terminates the current macro as though an .ENDM directive were encountered. Use of .MEXIT bypasses the complications of conditional nesting and alternate paths. For example:

```
1
2
3
4
5
6
7
8                               .MACRO  ALTR,N,M,R
9                               MOV     RT!N,RT!R       ;RESET PARAM
10                              .IF     EQ,N
11                              .MEXIT
12                              .ENDC
13                              MOV     RT!N,-(SP)      ;SET UP RETURN
14                              .ENDM
15
16 00046                        ALTR    0,1,2
   00046 016767                 MOV     RT0,RT2 ;RESET PARAM
         000014
         000016
                                .IF     EQ,0
                                .MEXIT
                                .ENDC
                                MOV     RT0,-(SP)       ;SET UP RETURN
17 00054                        ALTR    1,2,3
   00054 016767                 MOV     RT1,RT3 ;RESET PARAM
         000010
         000012
                                .IF     EQ,1
                                .MEXIT
                                .ENDC
   00062 016746                 MOV     RT1,-(SP)       ;SET UP RETURN
         000002
18
19 00066 000000 RT0:            .WORD   0
20 00070 000000 RT1:            .WORD   0
21 00072 000000 RT2:            .WORD   0
22 00074 000000 RT3:            .WORD   0
```

In an assembly where N=Ø, the .MEXIT directive terminates the macro expansion.

Where macros are nested, a .MEXIT causes an exit to the next higher level. A .MEXIT encountered outside a mcaro definition is flagged as an error.

6.1.4 MACRO Definition Formatting

A form feed character used as the only character on a line causes a page eject. Used within a macro definition, a form feed character causes a page eject. A page eject is not performed when the macro is invoked.

Used within a macro definition, the .PAGE directive is ignored, but a page eject is performed at invocation of that macro.

6.2 MACRO CALLS

A macro must be defined prior to its first reference. Macro calls are of the general form

```
[label:]    name[, real arguments]
```

where

label	represents an optional statement label.
name	represents the name of the macro specified in the .MACRO directive preceding the macro definition.
,	represents any legal separator (comma, space, or tab). No separator is necessary where there are no real arguments.
real arguments	are those symbols, expressions, and values that replace the dummy arguments in the .MACRO statement. Where more than one argument is used, they are separated by any legal separator.

Where a macro name is the same as a user label, the appearance of the symbol in the operation field designates a macro call, and the occurrence of the symbol in the operand field designates a label reference. For example:

```
1
2
3
4
5
6                       .MACRO  ABS,NUM
7                       .WORD   NUM
8                       .ENDM
9
10 000026 011011 ABS:   MOV     @R0,@R1         ;ABS AS A LABEL
11 000030 000776        BR      ABS             ;ABS AS AN OPERAND
12 000032         USE:
13 000032               ABS     4               ;ABS AS A MACRO CALL
   000032 000004        .WORD   4
14 000034 011011        MOV     @R0,@R1
```

Arguments to the macro call are treated as character strings whose usage is determined by the macro definition.

6.3 ARGUMENTS FOR MACRO CALLS AND DEFINITIONS

Arguments within a macro definition or macro call are separated from other arguments by any of the separating characters described in Section 6-3.1.1.

For example:

```
.MACRO      REN A,B,C
 .
 .
 .
REN         ALPHA,BETA,<C1,C2>
```

Arguments that contain separating characters are enclosed in paired angle brackets. An up-arrow construction is provided to allow angle brackets to be passed as arguments. Bracketed arguments are seldom used in a macro definition, but are more likely in a macro call. For example:

```
REN   <MOV X,Y>,#44,WEV
```

This call would cause the entire statement

```
MOV X,Y
```

to replace all occurrences of the symbol A in the macro definition. Real arguments within a macro call are considered to be character strings and are treated as a single entity until their use in the macro expansion.

The up-arrow construction could have been used in the above macro call as shown here.

```
REN   ↑/MOV X,Y/,#44,WEV
```

is equivalent to

```
REN   <MOV X,Y>,#44,WEV .
```

Since spaces are ignored preceding an argument, they can be used to increase legibility of bracketed constructions.

The form

```
REN  #44,WEV↑/MOV X,Y/
```

however, contains only two arguments, #44 and WEV↑/MOV X,Y/ (see Section 6-3.1.1), because ↑ is a unary operator.

6.3.1 Macro Nesting

Macro nesting (nested macro calls), where the expansion of one macro includes a call to another macro, causes one set of angle brackets to be removed from an argument with each nesting level. The depth of nesting allowed is dependent upon the amount of core space used by the program being assembled. To pass an argument containing

legal argument delimiters to nested macros, the argument should be enclosed in one set of angle brackets for each level of nesting, as shown below.

```
1
2
3
4
5
6
7
8
9                       .MACRO  LEVEL,FOO,FOO2
10                      LEVEL1  FOO
11                      LEVEL1  FOO2
12                      .ENDM
13
14                      .MACRO  LEVEL1,FOO3
15                      FOO3
16                      ADD     #10,R0          ;ADJUST TABLE POINTER
17                      MOV     R0,-(SP)        ;SAVE IT FOR LATER
18                      .ENDM
19
20 00076                LEVEL   <<MOV   RT0,R0>><<CLR   R1>>
   00076                LEVEL1  <MOV    RT0,R0>
   00076 016700         MOV     RT0,R0
         177764
   00102 062700         ADD     #10,R0          ;ADJUST TABLE POINTER
         000010
   00106 010046         MOV     R0,-(SP)        ;SAVE IT FOR LATER
   00110                LEVEL1  <CLR    R1>
   00110 005001         CLR     R1
   00112 062700         ADD     #10,R0          ;ADJUST TABLE POINTER
         000010
   00116 010046         MOV     R0,-(SP)        ;SAVE IT FOR LATER
21
```

Where macro definitions are nested (that is, a macro definition is entirely contained within the definition of another macro), the inner definition is not defined as a callable macro until the outer macro has been called and expanded. For example:

```
.MACRO  LV1 A,B
  .
  .
  .
.MACRO  LV2 A
  .
  .
  .
.ENDM
.ENDM
```

The LV2 macro cannot be called by name until after the first call to the LV1 macro. Likewise, any macro defined within the LV2 macro definition cannot be referenced directly until LV2 has been called.

6.3.2 Concatenation

The apostrophe or single quote character (') operates as a legal separating character in macro definitions. An ' character that precedes and/or follows a dummy argument in a macro definition is removed, and the substitution of the real argument occurs at that point. For example:

```
1
2
3
4
5
6
7                       .MACRO  DEF,I,J,K
8               RT'J:   .ASCIZ  /K/
9                       .EVEN
10                      .WORD   'I
11                      .ENDM
12
13 00142                DEF     RT5,6,<MACRO=11>
   00142    115 RT6:    .ASCIZ  /MACRO=11/
   00143    101
   00144    103
   00145    122
   00146    117
   00147    055
   00150    061
   00151    061
   00152    000
                        .EVEN
   00154 000000G        .WORD   RT5
```

Within nested macro definitions, multiple single quotes can be used, with one quote removed at each level of macro nesting.

6.3.3 Special Characters

Arguments may include special characters without enclosing the argument in a bracket construction, if that argument does not contain spaces, tabs, semicolons, or commas. For example:

```
.MACRO  PUSH ARG
MOV     ARG,-(SP)
.ENDM
 .
 .
 .
PUSH    X+3(%2)
```

generates the following code:

```
MOV     X+3(%2),-(SP)
```

6.3.4 Numeric Arguments Passed as Symbols

When passing macro arguments, a useful capability is to pass a symbol that can be treated by the macro as a numeric string. An argument preceded by the unary operator backslash (\) is treated as a number in the current radix. The ASCII characters representing the number are inserted in the macro expansion; their function is defined in context. For example:

```
1
2
3
4
5
6
7          000004           NUM=4
8          000000           COU=0
9                           .MACRO  INK,V1,V2
10                          COUNT   V1,◊V2
11                          COU=COU+1
12                          .ENDM
13
14                          .MACRO  COUNT,V1,V2
15                 RT'V1:   .WORD   V2
16                          .ENDM
17
18 00120                    INK     <10>,NUM
   00120                    COUNT   10,◊NUM
   00120 000004 RT10:       .WORD   4
         000001             COU=COU+1
```

Using this facility can make source listings somewhat clearer. For example, versions of programs created through conditional assembly of a single source can identify themselves as follows.

```
          .MACRO  IDT  SYM        ;ASSUME THAT THE SYMBOL ID TAKES
          .IDENT  /SYM/           ;ON A UNIQUE 2-DIGIT VALUE FOR
          .ENDM                   ;EACH POSSIBLE CONDITIONAL ASSEMBLY
          .MACRO  OUT ARG         ;OF THE PROGRAM
          IDT     ØØ5A'ARG            .
          .ENDM                       .
              .                       .
              .
              .
          OUT     \ID             ;WHERE ØØ5A IS THE UPDATE
                                  ;VERSION OF THE PROGRAM
                                  ;AND ARG INDICATES THE
                                  ;CONDITIONAL ASSEMBLY VERSION.
```

The above macro call expands to

```
          .IDENT   /ØØ5AXX/
```

where XX is the conditional value of ID.

Two macros are necessary since the text delimiting characters in the .IDENT statement would inhibit the concatenation of a dummy argument.

6.3.5 Number of Arguments

If more arguments appear in the macro call than in the macro definition, the excess arguments are ignored. If fewer arguments appear in the macro call than in the definition, missing arguments are assumed to be null (consist of no characters). The conditional directives .IF B and .IF NB can be used within the macro to detect unnecessary arguments.

A macro can be defined with no arguments.

6.3.6 Automatically Created Symbols

MACRO can create symbols of the form n$ where n is a decimal integer number such that 64<n<127. Created symbols are always local symbols between 64$ and 127$. (For a description of local symbols, see Section 6-3.7.) Such local symbols are created by the assembler in numerical order.

```
64$
65$
 .
 .
 .
126$
127$
```

Created symbols are particularly useful where a label is required in the expanded macro. Such a label must otherwise be explicitly stated as an argument with each macro call or the same label is generated with each expansion and results in a multi-defined label. Unless a label is referenced from outside the macro, there is no reason for the programmer to be concerned with that label.

The range of these local symbols extends between two explicit labels. Each new explicit label creates a new local symbol block.

The macro processor creates a local symbol on each call of a macro whose definition contains a dummy argument preceded by the ? (question mark) character. Local symbols are generated only where the real argument of the macro call is either null or missing. If a real argument is specified in the macro call, the generation of a local symbol is inhibited and normal replacement is performed. Consider the following example and expansions.

```
1
2
3
4
5
6
7                               .MACRO  TEST,REG,?LAB
8                               TST     REG
9                               BEQ     LAB
10                              ADD     #5,REG
11                      LAB:
12                              .ENDM
13
14  00122                       TEST    %1
    00122  005701               TST     %1
    00124  001402               BEQ     64$
    00126  062701               ADD     #5,%1
           000005
    00132                64$:
15  00132                       TEST    %1,XYZ
    00132  005701               TST     %1
    00134  001402               BEQ     XYZ
    00136  062701               ADD     #5,%1
           000005
    00142                XYZ:
```

These assembler-generated symbols are restricted to the first 16 (decimal) arguments of a macro definition.

6.4 .NARG, .NCHR, AND .NTYPE

These three directives allow the user to obtain the number of arguments in a macro call (.NARG), the number of characters in an argument (.NCHR), or the addressing mode of an argument (.NTYPE). Use of these directives permits selective modifications of a macro depending upon the nature of the arguments passed.

The .NARG directive enables the macro being expanded to determine the number of arguments supplied in the macro call.

Format:

```
        [label:]  .NARG   symbol
```

where

label — is an optional statement label.

symbol — is any legal symbol whose value is to be equated to the number of arguments in the macro call currently being expanded. The symbol can be used by itself or in expressions.

The .NARG directive can occur only within a macro definition.

```
1
2
3
4
5                               .MACRO  NOPP,NUM
6                               .NARG   SYM
7                               .IF     EQ,SYM
8                               .MEXIT
9                               .IFF
10                              .REPT   NUM
11                              NOP
12                              .ENDM
13                              .ENDC
14                              .ENDM
15
16 00202                        NOPP
          000000                .NARG   SYM
                                .IF     EQ,SYM
                                .MEXIT
                                .IFF
                                .REPT
                                NOP
                                .ENDM
                                .ENDC
17
18 00202                        NOPP    6
          000001                .NARG   SYM
                                .IF     EQ,SYM
                                .MEXIT
                                .IFF
          000006                .REPT   6
                                NOP
                                .ENDM
   00202 000240                 NOP
   00204 000240                 NOP
   00206 000240                 NOP
   00210 000240                 NOP
   00212 000240                 NOP
   00214 000240                 NOP
                                .ENDC
```

The .NCHR directive enables a program to determine the number of characters in a character string.

Format:

```
[label:]   .NCHR   symbol,  <character string>
```

where

label	is an optional statement label.
symbol	is any legal symbol that is to be equated to the number of characters in the specified character string. The symbol is separated from the character string argument by any legal separator.

<character string> is a string of printing characters that should only be enclosed in angle brackets if it contains a legal separator. A semicolon also terminates the character string.

The .NCHR directive can occur anywhere in a MACRO program.

```
1
2
3
4
5
6
7                               .MACRO  CHAR,MESS
8                               .NCHR   SYM,MESS
9                               .WORD   SYM
10                              .ASCII  /MESS/
11                              .ENDM
12
13 001720                       CHAR    <HELLO>
          000005                .NCHR   SYM,HELLO
   001722 000005                .WORD   SYM
   001724    110                .ASCII  /HELLO/
   001725    105
   001726    114
   001727    114
   001730    117
14                              .EVEN
```

The .NTYPE directive enables the macro being expanded to determine the addressing mode of any argument.

Format:

```
[label:]   .NTYPE   symbol, arg
```

where

label — is an optional statement label.

symbol — is any legal symbol, the value of which is to be equated to the 6-bit addressing mode of the argument. The symbol is separated from the argument by a legal separator. This symbol can be used by itself or in expressions.

arg — is any legal macro argument (dummy argument) as defined in Section 6-6.3.

The .NTYPE directive can occur only within a macro definition.

```
1
2
3
4
5
6
7                               .MACRO  SAVE,ARG
8                               .NTYPE  SYM,ARG
9                               .IF     EQ,SYM&70
10                              MOV     ARG,TEMP        ;REGISTER MODE
11                              .IFF
12                              MOV     #ARG,TEMP       ;NON-REGISTER MODE
13                              .ENDC
14                              .ENDM
15
16 00156 000000 TEMP:           .WORD   0
17
18 00160                        SAVE    %1
          000001                .NTYPE  SYM,%1
                                .IF     EQ,SYM&70
   00160 010167                 MOV     %1,TEMP ;REGISTER MODE
          177772 '
                                .IFF
                                MOV     #%1,TEMP        ;NON-REGISTER MODE
                                .ENDC
19
20 00164                        SAVE    TEMP
          000067                .NTYPE  SYM,TEMP
                                .IF     EQ,SYM&70
                                MOV     TEMP,TEMP       ;REGISTER MODE
                                .IFF
   00164 012767                 MOV     #TEMP,TEMP      ;NON-REGISTER MODE
          000156'
          177764
                                .ENDC
```

6.5 .ERROR AND .PRINT

The .ERROR directive is used to output messages to the command output device during assembly pass 2. A common use is to provide diagnostic announcements of a rejected or erroneous macro call. The form of the .ERROR directive is as follows:

```
[label:]    .ERROR  [expr];text
```

where

label	is an optional statement label.
expr	is an optional legal expression whose value is output to the command device when the .ERROR directive is encountered. Where expr is not specified, the text only is output to the command device.
;	denotes the beginning of the text string to be output.
text	is the one-line string to be output to the command device.

Upon encountering an .ERROR directive anywhere in a MACRO program, the assembler outputs a single line containing the following information.

1. The sequence number of the .ERROR directive line.
2. The current value of the location counter.
3. The value of the expression if one is specified.
4. The text string specified.

For example,

```
        .ERROR     A;UNACCEPTABLE MACRO ARGUMENT
```

causes a line similar to the following to be output:

```
        512  5642 ØØØØ76       ;UNACCEPTABLE MACRO ARGUMENT
```

where the above fields are, from left to right, sequence number, location counter, expression value, and text.

The line is flagged on the assembly listing with a P error code.

The .PRINT directive is identical to .ERROR except that it is not flagged with a P error code.

6.6 INDEFINITE REPEAT BLOCK: .IRP AND .IRPC

An indefinite repeat block is a structure very similar to a macro definition. An indefinite repeat is essentially a macro definition that has only one dummy argument and is expanded once for every real argument supplied. An indefinite repeat block is coded in line with its expansion, rather than being referenced by name as a macro is referenced.

An indefinite repeat block can occur either within or outside macro definitions, repeat ranges, or indefinite repeat ranges. The rules for creating an indefinite repeat block are the same as for the creation of a macro definition. Indefinite repeat arguments follow the same rules that apply to macro arguments.

Format:

```
        [label:]  .IRP arg,<real arguments>
                    .
                    .
                    .
                  (range of the indefinite repeat)
                    .
                    .
                    .
                  .ENDM
```

where

label	is an optional statement label. A label may not appear on any .IRP statement within another macro definition, repeat range or indefinite repeat range, or on any .ENDM statement.
arg	is a dummy argument that is successively replaced with the real arguments in the .IRP statement.
<real argument>	is a list of arguments to be used in the expansion of the indefinite repeat range and enclosed in angle brackets. Each real argument is a string of zero or more characters or a list of real arguments (enclosed in angle brackets). The real arguments are separated by commas.
range	is the block of code to be repeated once for each real argument in the list. The range may contain macro definitions, repeat ranges, or other indefinite repeat ranges. Note that only created symbols should be used as labels within an indefinite repeat range.

Figure 6-6 illustrates the use of .IRP.

```
1
2
3
4
5
6
7                               .IRP    X,<A,B,C>
8                               MOV     X,(R0)+
9                               .ENDM
  000216 016720                 MOV     A,(R0)+
         177556
  000222 016720                 MOV     B,(R0)+
         177554
  000226 016720                 MOV     C,(R0)+
         177552
```

Figure 6-6
.IRP Example

A second type of indefinite repeat block is available which handles character substitution rather than argument substitution. The .IRPC directive is used as follows:

```
label:  .IRPC arg,string
          .
          .
          .
        (range of indefinite repeat)
          .
          .
          .
        .ENDM
```

On each iteration of the indefinite repeat range, the dummy argument (arg) assumes the value of each successive character in the string.

6.7 REPEAT BLOCK: .REPT

Occasionally it is useful to duplicate the same block of code a number of times in line with other source code. This is performed by creating a repeat block of the following form.

```
[.label:]  .REPT expr
             .
             .
             .
           (range of repeat block)
             .
             .
             .
           .ENDM              ;OR .ENDR
```

where

label — is an optional statement label. The .ENDR or .ENDM directive may not have a label. A .REPT statement occurring within another repeat block, indefinite repeat block, or macro definition may not have a label associated with it.

expr — is any legal expression controlling the number of times the block of code is assembled. When the value of expr = Ø, the range of the repeat block is not assembled.

range — is the block of code to be repeated expr number of times. The range may contain macro definitions, conditionals, indefinite repeat ranges or other repeat ranges. Note that no statements within a repeat range can have a label.

The last statement in a repeat block can be an .ENDM or .ENDR statement. The .ENDR statement is provided for compatibility with previous assemblers.

The .MEXIT statement is also legal within the range of a repeat block.

```
1
2
3
4
5
6
7          000020          .REPT   20
8                          .WORD   0
9                          .ENDM
  000232 000000            .WORD   0
  000234 000000            .WORD   0
  000236 000000            .WORD   0
  000240 000000            .WORD   0
  000242 000000            .WORD   0
  000244 000000            .WORD   0
  000246 000000            .WORD   0
  000250 000000            .WORD   0
  000252 000000            .WORD   0
```

```
  000254 000000          .WORD   0
  000256 000000          .WORD   0
  000260 000000          .WORD   0
  000262 000000          .WORD   0
  000264 000000          .WORD   0
  000266 000000          .WORD   0
  000270 000000          .WORD   0
10        000001'        .END
```

6.8 MACRO LIBRARIES: .MCALL

All macro definitions must occur prior to their being referenced within the user program. MACRO provides a selection mechanism for the programmer to indicate in advance those system macro definitions required by his program.

The .MCALL directive is used to specify the names of all system macro definitions not defined in the current program but required by the program. The .MCALL directive must appear before the first occurrence of a macro call for an externally defined macro. The .MCALL directive has the following format.

```
.MCALL arg1[,arg2,...]
```

where arg1,arg2,... are the names of the macro definitions required in the current program.

When this directive is encountered, MACRO searches the system library SYSMAC.SML to find the requested definition(s).

PART 6

CHAPTER 7

OPERATING PROCEDURES

The MACRO Assembler assembles one or more ASCII source files containing MACRO statements. Its output consists of a relocatable binary object file and an assembly listing followed by the symbol table listing. A cross-reference listing (CREF) can be specified as part of the assembly output by means of a switch option.

7.1 LOADING MACRO

MACRO is loaded with the Disk Monitor RUN command.

```
$RUN MACRO
```

(Characters printed by the system are underlined to differentiate them from characters typed by the user.) The assembler responds by identifying itself and its version number, followed by a # character to indicate readiness to accept a command input string.

```
MACRO Vxxx
#
```

7.2 COMMAND INPUT STRING

In response to the # printed by the assembler, the user types the output file specification(s), a left angle bracket, and the input file specification(s).

Format:

```
#object,listing<sourcel,source2,...,sourceN
```

where

object	is the binary object file.
listing	is the assembly listing file containing the assembly listing and symbol table. Optionally, a separate CRF listing file can be appended to the assembly listing or output as a separate file.
sourcel,source2, ...,sourceN	are the ASCII source files containing the MACRO source program(s). No limit is set on the number of input source files, but the assembler is limited by the size of the user-defined and macro symbol tables.

A null specification in any of the file fields signifies that the associated input or output file is not desired. Each file specification contains the following information and follows the standard DOS conventions for file specifications.

```
dev:filnam.ext[uic]/option:arg
```

One or more switch options can be specified with each file specification to provide the assembler with information about that file. The switch options are described in Appendix J.

A syntactical error detected in the command string causes the assembler to output the command string up to and including the point where the error was detected, followed by a ? character. The assembler then reprints the # character and waits for a new command string to be entered. The following command string errors are detected.

Error	Error Message
Illegal switch Too many switches Illegal switch value Too many switch values	ILLEGAL SWITCH
Too many output file specifications	TOO MANY OUTPUT FILES
No input file specification	INPUT FILE MISSING

The default value for each file specification is noted below in Table 6-15.

Table 6-15
File Specification Default Values

	dev	filnam	ext	uic
object	system device	last source file name	.OBJ	current
listing	device used for object output	last source file name	.LST	current
CREF intermediate	system device	last source file name	.CRF	current
source1	system device	-	.MAC .PAL .null	current
source2 . . sourceN	device used for source1 (last source file specified)	-	.MAC .PAL .null	current

(continued on next page

Table 6-15 (Cont.)
File Specification Default Values

	dev	filnam	ext	uic
system macro file	system device	SYSMAC	.SML	current [1,1]

7.3 CROSS-REFERENCE TABLE GENERATION

A cross-reference listing (CREF) of all or a subset of all symbols used in the source program can be obtained by a call to the CREF routine. CREF can be used in two ways.

a. CREF can be called automatically following an assembly. In order to do this, the /CRF switch is specified following the assembly listing file specification. For example:

```
#,LP:/CRF<FILE1,FILE2
```

This command string sends the assembly listing (FILE2.LST) to the line printer. An intermediate CREF file is created and temporarily stored on the system device (FILE2.CRF) under the current UIC. The CREF routine takes this intermediate file, generates a CREF listing and routes that listing to the line printer. (The CREF listing is appended to the file FILE2.LST.) The CREF intermediate file is then deleted; there is no way to preserve this file when CREF is being called automatically.

b. If no CREF listing is desired immediately, the intermediate CREF file can be saved on the system device; the CREF listing can be generated at a later date. In order to preserve the intermediate CREF file, the following MACRO command string is given.

```
#,LP:/CRF:NG<FILE1,FILE2
```

This command string sends the assembly listing (FILE2.LST) to the line printer. The CREF intermediate file (FILE2.CRF) is sent to the system device under the current UIC. (The :NG argument is a mnemonic for "No Go" to CREF; i.e., no automatic transfer to the CREF routine following the output of the assembly listing.)

In order to generate the CREF listing, the CREF routine is run and given a command string indicating the input file specification(s) and a single output file specification. For example:

```
$RU
CREF VØØ1A
#LP:<FILE2.CRF
```

In this case the intermediate file created automatically in the MACRO example above is processed to obtain a CREF listing, which is then sent to the line printer. The CREF intermediate file is then automatically deleted. If it is desired to preserve the intermediate file, the following command string should be given.

```
#LP:<FILE2.CRF/SA
```

Unless the /SA switch is specified, the default case is always to delete the CREF intermediate file.

The CREF listing is organized into one to five sections, each listing a different type of symbol. The sections follow here.

Section Type	Argument
user-defined symbols	:S
macro symbolic names	:M
permanent symbols (instructions, directives)	:P
.PSECT symbolic names	:C
error codes	:E

Where no arguments are specified following the /CRF switch, all of the above sections except the permanent symbols are cross-referenced. However, if any one argument is specified (other than :NG), then no other default sections are assumed or provided. For example, in order to obtain a CREF listing for all five section types, the following switch option specification is used.

```
/CRF:S:M:P:C:E
```

The order in which the arguments are specified does not affect the order of their output, as is listed above.

Figure 6-7 contains a segment of source code and Figure 6-8 contains a segment of a CREF listing with some references to the code in Figure 6-7.

In a CREF listing, each cross-referenced symbol is printed in the left-hand column, followed by a list of the page-line numbers of the places at which that symbol is referenced. A # character following a page-line number indicates the point at which the listed symbol is defined. An @ character designates a page-line number at which the contents of that symbol are possibly altered.

```
1                               .SBTTL  OBJECT CODE HANDLERS
2
3 Ø12Ø26                ENDP:                                   ;END OF PASS HANDLER
4 Ø12Ø26                        CALL    SETMAX
  Ø12Ø26 ØØ4767                 JSR     PC,SETMAX
         17424Ø
5 Ø12Ø32 ØØ5767                 TST     PASS                    ;PASS ONE?
         ØØØØØØ'
6 Ø12Ø36 ØØ1142                 BNE     ENDP2                   ;BRANCH IF PASS 2
7 Ø12Ø4Ø                        ENTOVR  4
8 Ø12Ø4Ø ØØ5767                 TST     OBJLNK                  ;PASS ONE, ANY OBJECT?
         ØØ1416'
9 Ø12Ø44 ØØ1517                 BEQ     3Ø$                     ;  NO
1Ø 12Ø46 Ø12767                 MOV     #BLKTØ1,BLKTYP          ;SET BLOCK TYP1 1
         ØØØØØ1
         ØØØ542'
11 12Ø54                        CALL    OBJINI                  ;INIT THE POINTERS
   12Ø54 ØØ4767                 JSR     PC,OBJINI
         ØØ1542
12 12Ø6Ø Ø127Ø1                 MOV     #PRGTTL,R1              ;SET "FROM" INDEX
         ØØØØ5Ø'
13 12Ø64 Ø167Ø2                 MOV     RLDPNT,R2               ;  AND "TO" INDEX
         ØØØ54Ø'
14 12Ø7Ø                        CALL    GSDDMP                  ;OUTPUT GSD BLOCK
   12Ø7Ø ØØ4767                 JSR     PC,GSDDMP
         ØØØ66Ø
15 12Ø74 ØØ5Ø46                 CLR     -(SP)                   ;INIT FOR SECTOR SCAN
16 12Ø76 Ø12667         1Ø$:    MOV     (SP)+,ROLUPD            ;SET SCAN MARKER
         ØØØØØ6'
17 121Ø2                        NEXT    SECROL                  ;GET THE NEXT SECTOR
   121Ø2 Ø127ØØ                 MOV     #SECROL,RØ
         ØØØØ1Ø
   121Ø6 ØØ4767                 JSR     PC,NEXT
         ØØ54ØØ
18 12112 ØØ145Ø                 BEQ     2Ø$                     ;BRANCH IF THROUGH
19 12114 Ø16746                 MOV     ROLUPD,-(SP)            ;SAVE MARKER
         ØØØØØ6'
2Ø 1212Ø Ø127Ø1                 MOV     #MODE,R1
         ØØØØØ6'
21 12124 Ø111Ø5                 MOV     (R1),R5                 ;SAVE SECTOR
22 12126 Ø427Ø5                 BIC     #377,R5                 ;ISOLATE IT
         ØØØ377
23 12132 ØØØ3Ø5                 SWAB    R5                      ;  AND PLACE IN RIGHT
24 12134 Ø42711                 BIC     #-1-<RELFLG>,(R1)       ;CLEAR ALL BUT REL BIT
         177737
25 1214Ø Ø52721                 BIS     #<GSDTØ1>+DEFFLG.(R1)+  ;SET TO TYPE 1, DEFINED
         ØØØ41Ø
26 12144 Ø1Ø521                 MOV     R5,(R1)+                ;ASSURE ABS
27 12146 ØØ14Ø1                 BEQ     11$                     ;  OOPS!
28 1215Ø Ø11141                 MOV     (R1),-(R1)              ;  REL, SET MAX
29 12152 ØØ5Ø67         11$:    CLR     ROLUPD                  ;SET FOR INNER SCAN
         ØØØØØ6'
3Ø 12156 Ø127Ø1         12$:    MOV     #SYMBOL,R1
         ØØØØØ2'
31 12162                        CALL    GSDDMP                  ;OUTPUT THIS BLOCK
   12162 ØØ4767                 JSR     PC,GSDDMP
         ØØØ56
```

Figure 6-7
Assembly Listing

```
32 12166          13$:    NEXT    SYMBOL              ;FETCH THE NEXT SYMBOL
   12166 Ø127ØØ           MOV     #SYMBOL.RØ
         ØØØØØØ
   12172 ØØ4767           JSR     PC,NEXT
         ØØ5314
33 12176 ØØ1737           BEQ     1Ø$                 ;  FINISHED WITH THIS GUY
34 122ØØ Ø32767           BIT     #GLBFLG,MODE        ;GLOBAL?
         ØØØ1ØØ
         ØØØØØ6'
35 122Ø6 ØØ1767           BEQ     13$                 ;  NO
36 1221Ø 1267Ø5           CMPB    SECTOR,R5           ;YES, PROPER SECTOR?
         ØØØØØ7'
37 12214 ØØ1364           BNE     13$                 ;  NO
38 12216 Ø42767           BIC     #-1-<DEFFLG!RELFLG!GLBFLG>,MODE  ;CLEAR MOST
         177627
         ØØØØØ6'
39 12224 Ø52767           BIS     #GSDTØ4,MODE        ;SET TYPE 4
         ØØ2ØØØ
         ØØØØØ6'
4Ø 12232 ØØØ751           BR      12$                 ;OUTPUT IT
```

Figure 6-7 (cont.)
Assembly Listing

```
ENDMAC    27-4Ø    1Ø9-33#
ENDP      23-23     72- 3#
ENDP1M    73-16     72-22#
ENDP2     72- 6     74- 1#
  .
  .
  .
MDFFLG    12- 7#    35-28     92- 8     92-24
MEXIT    116- 1#   116-41#
MODE      14- 6#    22-29@    34-12     35-17@    36-12     37- 4     4Ø-43
          45- 6@    48-16@    58-38@    64-23     7Ø-1Ø     72-2Ø     72-34
          72-38@    72-39@    74-34     75-37     86- 8     91-2Ø@   1Ø6-27
         116-34#
MOVBYT    18- 5     18- 9     28-44     74-41     83-11     83-2Ø    1Ø8-19#
MPDP     1Ø9-42    121-17#
MPUSH    1Ø9-26    11Ø-48    121- 1#
MSBARG    27- 9    121-18    121-4Ø#
MSBBLK   121- 4    121-28    121-36#
MSBCNT    27-15    1Ø9-33    116- 6    121-41#
MSBEND   121- 9    121-28    121-43#
MSBMRP    25-19     27-25@   11Ø-49@   121-42#
```

Figure 6-8
Excerpts from CREF Listing to Accompany Figure 6-7

Note particularly the CREF references for ENDP, ENDP2, and MODE.

PART 6

CHAPTER 8

A SUMMARY OF THE MACRO ASSEMBLY LANGUAGE AND ASSEMBLER

8.1 SPECIAL CHARACTERS

Character	Function
vertical tab	Source line terminator
:	Label terminator
=	Direct assignment indicator
%	Register term indicator
tab	Item terminator
	Field terminator
space	Item terminator
	Field terminator
#	Immediate expression indicator
@	Deferred addressing indicator
(	Initial register indicator
)	Terminal register indicator
, (comma)	Operand field separator
;	Comment field indicator
+	Arithmetic addition operator or auto increment indicator
-	Arithmetic subtraction operator or auto decrement indicator
*	Arithmetic multiplication operator
/	Arithmetic division operator
&	Logical AND operator
!	Logical OR operator
"	Double ASCII character indicator
' (apostrophe)	Single ASCII character indicator
.	Assembly location counter
<	Initial argument indicator
>	Terminal argument indicator
↑	Universal unary operator
\	Argument indicator MACRO numeric argument indicator
::	Global label terminator
==	Global assignment indicator

8.2 ADDRESS MODE SYNTAX

n is an integer between Ø and 7 representing a register. R is a register expression, E is an expression, ER is either a register expression or an expression in the range Ø to 7.

Format	Address Mode Name	Address Mode Number	Meaning
R	register	Øn	Register R contains the operand. R is a register expression.
@R or (ER)	deferred register	1n	Register R contains the operand address.
(ER)+	autoincrement	2n	The contents of the register specified by ER are incremented after being used as the address of the operand.
@(ER)+	deferred auto-increment	3n	ER contains the pointer to the address of the operand. ER is incremented after use.
-(ER)	autodecrement	4n	The contents of register ER are decremented before being used as the address of the operand.
@-(ER)	deferred auto-decrement	5n	The contents of register ER are decremented before being used as the pointer to the address of the operand.
E(ER)	index	6n	E plus the contents of the register specified, ER, is the address of the operand.
@E(ER)	deferred index	7n	E plus the contents of the register specified, ER, is the address of the address of the operand.
#E	immediate	27	E is the operand.
@#E	absolute	37	E is the address of the operand.
E	relative	67	E is the address of the operand.
@E	deferred relative	77	E is the pointer to the address of the operand.

Form	Described In Section	Operation
'	6-5.3.3	A single quote character (apostrophe) followed by one ASCII character generates a word contains the 7-bit ASCII representation of the character in the low-order byte, and zero in the high-order byte.
	6-6.3.2	Concatenation within a macro.
"	6-5.3.3	A double quote character followed by two ASCII characters generates a word containing the 7-bit ASCII representation of the two characters.
↑Bn	6-5.4.2	Temporary radix control; causes the number n to be treated as a binary number.
↑Cn	6-5.6.2	Creates a word containing the one's complement of n.
↑Dn	6-5.4.2	Temporary radix control; causes the number n to be treated as a decimal number.
↑Fn	6-5.6.2	Creates a one-word floating point quantity to represent n.
↑On	6-5.4.2	Temporary radix control; causes the number n to be treated as an octal number.
.ASCII string	6-5.3.4	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters) one character per byte.
.ASCIZ string	6-5.3.5	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters), one character per byte with a zero byte following the specified string.
.ASECT	6-5.9.2	Begin or resume absolute section.
.BLKB exp	6-5.5.3	Reserves a block of storage space exp bytes long.
.BLKW exp	6-5.5.3	Reserves a block of storage space exp words long.
.BYTE expl,exp2,...	6-5.3.1	Generates successive bytes of data containing the octal equivalent of the expression(s) specified.
.CSECT symbol	6-5.9.2	Begin or resume named or unnamed relocatable section.

(continued on next page)

Form	Described In Section	Operation
.DSABL arg	6-5.2	Disables the assembler function specified by the argument.
.ENABL arg	6-5.2	Provides the assembler function specified by the argument.
.END .END exp	6-5.7.1	Indicates the physical end of source program. An optional argument specifies the transfer address.
.ENDC	6-5.11	Indicates the end of a conditional block.
.ENDM .ENDM symbol	6-6.1.2	Indicates the end of the current repeat block, indefinite repeat block, or macro. The optional symbol, if used, must be identical to the macro name.
.EOT	6-5.7.2	Ignored. Indicates end-of-tape, which is detected automatically by the hardware.
.ERROR exp,string	6-6.5	Causes a text string containing the optional expression and the indicated text string to be output to the command device.
.EVEN	6-5.5.1	Ensures that the assembly location counter contains an even address by adding 1 if it is odd.
.FLT2 arg1,arg2,...	6-5.6.1	Generates successive two-word floating-point equivalents for the floating-point numbers specified as arguments.
.FLT4 arg1,arg2,...	6-5.6.1	Generates successive four-word floating-point equivalents for the floating-point numbers specified as arguments.
.GLOBL sym1,sym2,...	6-5.10	Defines the symbol(s) specified as global symbol(s).
.IDENT symbol	6-5.1.5	Provides a means of labeling the object module with the program version number. The symbol is the version number between paired delimiting characters.
.IF cond,arg1,arg2,...	6-5.11	Begins a conditional block of source code, which is included in the assembly only if the stated condition is met with respect to the argument(s) specified.
.IFF	6-5.11.1	Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested false.

(continued on next page)

Form	Described In Section	Operation
.IFT	6-5.11.1	Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested true.
.IFTF	6-5.11.1	Appears only within a conditional block and indicates the beginning of a section of code to be unconditionally assembled.
.IIF cond,arg,statement	6-5.11.2	Acts as a one-line conditional block where the condition is tested for the argument specified. The statement is assembled only if the condition tests true.
.IRP sym,<arg1,arg2,...>	6-6.6	Indicates the beginning of an indefinite repeat block in which the symbol specified is replaced with successive elements of the real argument list (which is enclosed in angle brackets).
.IRPC sym,string	6-6.6	Indicates the beginning of an indefinite repeat block in which the symbol specified takes on the value of successive characters in the character string.
.LIMIT	6-5.8	Reserves two words into which the Linker inserts the low and high addresses of the relocated code.
.LIST .LIST arg	6-5.1.1	Without an argument, .LIST increments the listing level count by 1. With an argument, .LIST does not alter the listing level count but formats the assembly listing according to the argument specified.
.MACRO sym,arg1,arg2,...	6-6.1.1	Indicates the start of a macro named sym containing the dummy arguments specified.
.MEXIT	6-6.1.3	Causes an exit from the current macro or indefinite repeat block.
.NARG symbol	6-6.4	Appears only within a macro definition and equates the specified symbol to the number of arguments in the macro call currently being expanded.
.NCHR sym,string	6-6.4	Can appear anywhere in a source program; equates the symbol specified to the number of characters in the string (enclosed in delimiting characters).
.NLIST .NLIST arg	6-5.1.1	Without an argument, .NLIST decrements the listing level count by 1. With an argument, .NLIST deletes the portion of the listing indicated by the argument.

(continued on next page)

Form	Described In Section	Operation
.NTYPE sym,arg	6-6.4	Appears only in a macro definition and sets the low-order six bits of the symbol specified to the six-bit addressing mode of the argument.
.ODD	6-5.5.2	Ensures that the assembly location counter contains an odd address by adding 1 if it is even.
.PAGE	6-5.1.6	Causes the assembly listing to skip to the top of the next page.
.PSECT	6-5.9	Begin or resume a program section.
.PRINT exp,string	6-6.5	Causes a text string to be output to the command device containing the optional expression specified and the indicated text string.
.RADIX n	6-5.4.1	Alters the current program radix to n, where n can be 2, 4, 8, or 1Ø.
.RAD5Ø string	6-5.3.6	Generates a block of data containing the Radix-5Ø equivalent of the character string (enclosed in delimiting characters).
.REPT exp	6-6.7	Begins a repeat block. Causes the section of code up to the next .ENDM or .ENDR to be repeated exp times.
.SBTTL string	6-5.1.4	Causes the string to be printed as part of the assembly listing page header. The string part of each .SBTTL directive is collected into a table of contents at the beginning of the assembly listing.
.TITLE string	6-5.1.3	Assigns the first symbolic name in the string to the object module and causes the string to appear on each page of the assembly listing. One .TITLE directive should be issued per program.
.WORD expl,exp2,...	6-5.3.2	Generates successive words of data containing the octal equivalent of the expression(s) specified.

PART 6

CHAPTER 9

PERMANENT SYMBOL TABLE

The Permanent Symbol Table (PST) defines values for each symbol that is automatically recognized by MACRO. The symbols defined include op-codes and macro-calls. A listing of the Permanent Symbol Table forms the balance of this chapter.

```
;
; PERMANENT SYMBOL TABLE
;
; EQUATED SYMBOLS
;

DR1=    200                     ;DESTRUCTIVE REFERENCE IN FIRST FIELD
DR2=    100                     ;DESTRUCTIVE REFERENCE IN SECOND FIELD

DFLGEV==020                     ;DIRECTIVE REQUIRES EVEN LOCATION
DFLGBM==010                     ;DIRECTIVE USES BYTE MODE
DFLCND==004                     ;CONDITIONAL DIRECTIVE
DFLMAC==002                     ;MACRO DIRECTIVE
DFLSMC==001                     ;MCALL

;
; LOCAL MACROS
;

        .IF DF  PAL11R          ;PAL11R SUBSET
XMACRO= 0
X40=    0
X45=    0
        .ENDC

        .IIF DF X40&X45,        XFLTG=  0
        .IIF DF XMACRO, XSML=   0

        .MACRO  OPCDEF  NAME,   CLASS,  VALUE,  FLAGS,  COND
        .IF NB  <COND>
        .IF DF  COND
        .MEXIT
        .ENDC
        .ENDC
        .RAD50  /NAME/
        .BYTE   FLAGS+0
        .BYTE   200+OPCL'CLASS
        .WORD   VALUE
        .ENDM

        .MACRO  DIRDEF  NAME,   FLAGS,  COND
        .RAD50  /.'NAME/
        .BYTE   FLAGS+0,        0
        .IF NB  <COND>
        .IF DF  COND
```

```
        .WORD   OPCERR
        .MEXIT
        .ENDC
        .ENDC
        .WORD   NAME
        .ENDM

        .MACRO  DIRDF1 NAME,ENTRY,FLAGS,COND
        .RAD50  /.'NAME/
        .BYTE   FLAGS,0
        .IF NB  <COND>
        .IF DF  COND
        .WORD   OPCERR
        .MEXIT
        .ENDC
        .ENDC
        .WORD   ENTRY
        .ENDM

PSTBAS::                                ;REF LABEL
        OPCDEF  <ABSD  >,       01,     170600, DR1,    X45
        OPCDEF  <ABSF  >,       01,     170600, DR1,    X45
        OPCDEF  <ADC   >,       01,     005500, DR1
        OPCDEF  <ADCB  >,       01,     105500, DR1
        OPCDEF  <ADD   >,       02,     060000, DR2
        OPCDEF  <ADDD  >,       11,     172000, DR2,    X45
        OPCDEF  <ADDF  >,       11,     172000, DR2,    X45
        OPCDEF  <ASH   >,       09,     072000, DR2,    X40&X45
        OPCDEF  <ASHC  >,       09,     073000, DR2,    X40&X45
        OPCDEF  <ASL   >,       01,     006300, DR1
        OPCDEF  <ASLB  >,       01,     106300, DR1
        OPCDEF  <ASR   >,       01,     006200, DR1
        OPCDEF  <ASRB  >,       01,     106200, DR1
        OPCDEF  <BCC   >,       04,     103000,
        OPCDEF  <BCS   >,       04,     103400,
        OPCDEF  <BEQ   >,       04,     001400,
        OPCDEF  <BGE   >,       04,     002000,
        OPCDEF  <BGT   >,       04,     003000,
        OPCDEF  <BHI   >,       04,     101000,
        OPCDEF  <BHIS  >,       04,     103000,
        OPCDEF  <BIC   >,       02,     040000, DR2
        OPCDEF  <BICB  >,       02,     140000, DR2
        OPCDEF  <BIS   >,       02,     050000, DR2
        OPCDEF  <BISB  >,       02,     150000, DR2
        OPCDEF  <BIT   >,       02,     030000,
        OPCDEF  <BITB  >,       02,     130000,
        OPCDEF  <BLE   >,       04,     003400,
        OPCDEF  <BLO   >,       04,     103400,
        OPCDEF  <BLOS  >,       04,     101400,
        OPCDEF  <BLT   >,       04,     002400,
        OPCDEF  <BMI   >,       04,     100400,
        OPCDEF  <BNE   >,       04,     001000,
        OPCDEF  <BPL   >,       04,     100000,
        OPCDEF  <BPT   >,       00,     000003,     .   X45
        OPCDEF  <BR    >,       04,     000400,
        OPCDEF  <BVC   >,       04,     102000,
        OPCDEF  <BVS   >,       04,     102400,
        OPCDEF  <CCC   >,       00,     000257,
        OPCDEF  <CFCC  >,       00,     170000,     .   X45
        OPCDEF  <CLC   >,       00,     000241,
        OPCDEF  <CLN   >,       00,     000250,
        OPCDEF  <CLR   >,       01,     005000, DR1
        OPCDEF  <CLRB  >,       01,     105000, DR1
```

```
OPCDEF  <CLRD  >,       01,     170400, DR1.    X45
OPCDEF  <CLRF  >,       01,     170400, DR1.    X45
OPCDEF  <CLV   >,       00,     000242,
OPCDEF  <CLZ   >,       00,     000244,
OPCDEF  <CMP   >,       02,     020000,
OPCDEF  <CMPB  >,       02,     120000,
OPCDEF  <CMPD  >,       11,     173400,     .   X45
OPCDEF  <CMPF  >,       11,     173400,     .   X45
OPCDEF  <CNZ   >,       00,     000254,
OPCDEF  <COM   >,       01,     005100, DR1
OPCDEF  <COMB  >,       01,     105100, DR1
OPCDEF  <DEC   >,       01,     005300, DR1
OPCDEF  <DECB  >,       01,     105300, DR1
OPCDEF  <DIV   >,       07,     071000, DR2.    X40&X45
OPCDEF  <DIVD  >,       11,     174400, DR2.    X45
OPCDEF  <DIVF  >,       11,     174400, DR2.    X45
OPCDEF  <EMT   >,       06,     104000,
OPCDEF  <FADD  >,       03,     075000, DR1.    X40
OPCDEF  <FDIV  >,       03,     075030, DR1.    X40
OPCDEF  <FMUL  >,       03,     075020, DR1.    X40
OPCDEF  <FSUB  >,       03,     075010, DR1.    X40
OPCDEF  <HALT  >,       00,     000000,
OPCDEF  <INC   >,       01,     005200, DR1
OPCDEF  <INCB  >,       01,     105200, DR1
OPCDEF  <IOT   >,       00,     000004,
OPCDEF  <JMP   >,       01,     000100,
OPCDEF  <JSR   >,       05,     004000, DR1
OPCDEF  <LDCDF >,       11,     177400, DR2.    X45
OPCDEF  <LDCFD >,       11,     177400, DR2.    X45
OPCDEF  <LDCID >,       14,     177000, DR2.    X45
OPCDEF  <LDCIF >,       14,     177000, DR2.    X45
OPCDEF  <LDCLD >,       14,     177000, DR2.    X45
OPCDEF  <LDCLF >,       14,     177000, DR2.    X45
OPCDEF  <LDD   >,       11,     172400, DR2.    X45
OPCDEF  <LDEXP >,       14,     176400, DR2.    X45
OPCDEF  <LDF   >,       11,     172400, DR2.    X45
OPCDEF  <LDFPS >,       01,     170100,     .   X45
OPCDEF  <LDSC  >,       00,     170004,     .   X45
OPCDEF  <LDUB  >,       00,     170003,     .   X45
OPCDEF  <MARK  >,       10,     006400,     .   X45
OPCDEF  <MFPD  >,       01,     106500,     .   X45
OPCDEF  <MFPI  >,       01,     006500,     .   X45
OPCDEF  <MODD  >,       11,     171400, DR2.    X45
OPCDEF  <MODF  >,       11,     171400, DR2.    X45
OPCDEF  <MOV   >,       02,     010000, DR2
OPCDEF  <MOVB  >,       02,     110000, DR2
OPCDEF  <MTPD  >,       01,     106600, DR1.    X45
OPCDEF  <MTPI  >,       01,     006600, DR1.    X45
OPCDEF  <MUL   >,       07,     070000, DR2.    X40&X45
OPCDEF  <MULD  >,       11,     171000, DR2.    X45
OPCDEF  <MULF  >,       11,     171000, DR2.    X45
OPCDEF  <NEG   >,       01,     005400, DR1
OPCDEF  <NEGB  >,       01,     105400, DR1
OPCDEF  <NEGD  >,       01,     170700, DR1.    X45
OPCDEF  <NEGF  >,       01,     170700, DR1.    X45
OPCDEF  <NOP   >,       00,     000240,
OPCDEF  <RESET >,       00,     000005,
OPCDEF  <ROL   >,       01,     006100, DR1
OPCDEF  <ROLB  >,       01,     106100, DR1
OPCDEF  <ROR   >,       01,     006000, DR1
OPCDEF  <RORB  >,       01,     106000, DR1
OPCDEF  <RTI   >,       00,     000002,
OPCDEF  <RTS   >,       03,     000200, DR1
```

```
OPCDEF  <RTT    >,      00,     000006,     .       X45
OPCDEF  <SBC    >,      01,     005600,  DR1
OPCDEF  <SBCB   >,      01,     105600,  DR1
OPCDEF  <SCC    >,      00,     000277,
OPCDEF  <SEC    >,      00,     000261,
OPCDEF  <SEN    >,      00,     000270,
OPCDEF  <SETD   >,      00,     170011,     .       X45
OPCDEF  <SETF   >,      00,     170001,     .       X45
OPCDEF  <SETI   >,      00,     170002,     .       X45
OPCDEF  <SETL   >,      00,     170012,     .       X45
OPCDEF  <SEV    >,      00,     000262,
OPCDEF  <SEZ    >,      00,     000264,
OPCDEF  <SOB    >,      08,     077000,  DR1.       X45
OPCDEF  <SPL    >,      13,     000230,     .       X45
OPCDEF  <STA0   >,      00,     170005,     .       X45
OPCDEF  <STB0   >,      00,     170006,     .       X45
OPCDEF  <STCDF  >,      12,     176000,  DR2.       X45
OPCDEF  <STCDI  >,      12,     175400,  DR2.       X45
OPCDEF  <STCDL  >,      12,     175400,  DR2.       X45
OPCDEF  <STCFD  >,      12,     176000,  DR2.       X45
OPCDEF  <STCFI  >,      12,     175400,  DR2.       X45
OPCDEF  <STCFL  >,      12,     175400,  DR2.       X45
OPCDEF  <STD    >,      12,     174000,  DR2.       X45
OPCDEF  <STEXP  >,      12,     175000,  DR2.       X45
OPCDEF  <STF    >,      12,     174000,  DR2.       X45
OPCDEF  <STFPS  >,      01,     170200,  DR1.       X45
OPCDEF  <STQ0   >,      00,     170007,     .       X45
OPCDEF  <STST   >,      01,     170300,  DR1.       X45
OPCDEF  <SUB    >,      02,     160000,  DR2
OPCDEF  <SUBD   >,      11,     173000,  DR2.       X45
OPCDEF  <SUBF   >,      11,     173000,  DR2.       X45
OPCDEF  <SWAB   >,      01,     000300,  DR1
OPCDEF  <SXT    >,      01,     006700,  DR1.       X45
OPCDEF  <TRAP   >,      06,     104400,
OPCDEF  <TST    >,      01,     005700,
OPCDEF  <TSTB   >,      01,     105700,
OPCDEF  <TSTD   >,      01,     170500,     .       X45
OPCDEF  <TSTF   >,      01,     170500,     .       X45
OPCDEF  <WAIT   >,      00,     000001,
OPCDEF  <XOR    >,      05,     074000,  DR2.       X45
DIRDEF  <ASCII>.        DFLGRM
DIRDEF  <ASCIZ>.        DFLGRM
DIRDEF  <ASECT>.        ,       XREL
DIRDEF  <BLKB >
DIRDEF  <BLKW >.        DFLGEV
DIRDEF  <BYTE >.        DFLGRM
DIRDEF  <CSECT>.        ,       XREL
.IF DF  YPHASE
DIRDEF  <DEPHA>
.ENDC
DIRDEF  <DSABL>
DIRDEF  <ENABL>
DIRDEF  <END  >
DIRDEF  <ENDC >.        DFLCND
DIRDF1  <ENDM >,ENDM,DFLMAC,XMACRO
DIRDF1  <ENDR >,ENDM,DFLMAC,XMACRO
DIRDEF  <EOT  >
DIRDEF  <ERROR>
DIRDEF  <EVEN >
DIRDEF  <FLT2 >.        DFLGEV, XFLTG
DIRDEF  <FLT4 >.        DFLGEV, XFLTG
DIRDEF  <GLOBL>.        ,       XREL
DIRDEF  <IDENT>
```

```
        DIRDEF  <IF    >,          DFLCND
        DIRDF1  <IFDF  >,IFDF,DFLCND
        DIRDF1  <IFEQ  >,IFDF,DFLCND
        DIRDEF  <IFF   >,          DFLCND
        DIRDF1  <IFG   >,IFDF,DFLCND
        DIRDF1  <IFGE  >,IFDF,DFLCND
        DIRDF1  <IFGT  >,IFDF,DFLCND
        DIRDF1  <IFL   >,IFDF,DFLCND
        DIRDF1  <IFLE  >,IFDF,DFLCND
        DIRDF1  <IFLT  >,IFDF,DFLCND
        DIRDF1  <IFNDF>,IFDF,DFLCND
        DIRDF1  <IFNE  >,IFDF,DFLCND
        DIRDF1  <IFNZ  >,IFDF,DFLCND
        DIRDEF  <IFT   >,          DFLCND
        DIRDEF  <IFTF  >,          DFLCND
        DIRDF1  <IFZ   >,IFDF,DFLCND
        DIRDEF  <IIF   >
        DIRDEF  <IRP   >,          DFLMAC, XMACRO
        DIRDEF  <IRPC  >,          DFLMAC, XMACRO
        DIRDEF  <LIMIT>,           DFLGFV, XREL
        DIRDEF  <LIST  >
        DIRDF1  <MACR  >,MACR,DFLMAC,XMACRO
        DIRDF1  <MACRO>,MACR,DFLMAC,XMACRO
        DIRDEF  <MCALL>,           DFLSMC, XSML
        DIRDEF  <MEXIT> ,          ,       XMACRO
        DIRDEF  <NARG  > ,         ,       XMACRO
        DIRDEF  <NCHR  > ,         ,       XMACRO
        DIRDEF  <NLIST>
        DIRDEF  <NTYPE> ,          ,       XMACRO
        DIRDEF  <ODD   >
        DIRDEF  <PAGE  >
        .IF DF  YPHASE
        DIRDEF  <PHASE>
        .ENDC
        DIRDEF  <PRINT>
        DIRDEF  <PSECT>
        DIRDEF  <RADIX>
        DIRDEF  <RAD50>,           DFLGFV
        DIRDEF  <REM   >
        DIRDEF  <REPT  >,          DFLMAC, XMACRO
        DIRDEF  <SBTTL>
        DIRDEF  <TITLE>
WRDSYM::                           ;REF LABEL
        DIRDEF  <WORD  >,          DFLGFV
PSTTOP::                           ;REF LABEL

        .END
```

PART 6

CHAPTER 10

WRITING POSITION-INDEPENDENT CODE

All addressing modes involving only register references are position-independent. These modes are as follows.

```
  R      register mode
 @R      deferred register mode
 (R)+    autoincrement mode
@(R)+    deferred autoincrement mode
-(R)     autodecrement mode
@-(R)    deferred autodecrement mode
```

When using these addressing modes, position-independence is guaranteed providing the contents of the registers have been supplied such that they are not dependent upon a particular core location.

The relative addressing modes are generally position-independent. These modes follow.

```
 A       relative mode
@A       relative deferred mode
```

Relative modes are not position-independent when A is an absolute address (that is, a nonrelocatable address) that is referenced from a relocatable module.

Index modes can be either position-independent or nonposition-independent, according to their use in the program. These modes follow here.

```
 X(R)    index mode
@X(R)    index deferred mode
```

If the base X is position-independent, the reference is also position-independent. For example:

```
        MOV    2(SP),RØ          ;POSITION-INDEPENDENT

        N=4
        MOV    N(SP),RØ          ;POSITION-INDEPENDENT

ADDR:   CLR    ADDR(R1)          :NONPOSITION-INDEPENDENT
```

Caution must be exercised in the use of index modes in position-independent code.

Immediate mode can also be either position-independent or not, according to its usage. Immediate mode references are formatted as shown here.

```
 #N      immediate mode
```

where N is an absolute number or a symbol defined by an absolute direct assignment, the code is position-independent. When a label replaces N, the code is nonposition-independent. (That is, immediate mode references are position-independent only where N is an absolute value.)

Absolute mode addressing is unlikely to be position-independent and should be avoided when coding position-independently. Absolute mode addressing references are formatted as shown here.

```
@#A     absolute mode
```

Since this mode is used to obtain the contents of a specific core address, it violates the intentions of position-independent code.

Such a reference is position-independent if A is an absolute address.

Position-independent code is used in writing programs such as device drivers and utility routines, which are most useful when they can be brought into any available core space. Figure 6-9 and Figure 6-10 show pieces of device driver code; one is position-independent and one is not.

```
; DVRINT -- ADDRESS OF DEVICE DRIVER INTERRUPT SERVICE
; VECTOR -- ABSOLUTE ADDRESS OF DEVICE INTERRUPT VECTOR
; DRIVER -- START ADDRESS OF DEVICE DRIVER

        MOV     #DVRINT,VECTOR      ;SET INTERRUPT ADDRESS
MOVB            DRIVER+6,VECTOR+2   ;SET PRIORITY
        CLRB    VECTOR+3            ;CLEAR UPPER STATUS BYTE
```

Figure 6-9
Nonposition-independent Code

```
        MOV     PC,R1               ;GET DRIVER START
        ADD     #DRIVER-.,R1
        MOV     #VECTOR,R2          ;...& VECTOR ADDRESSES
        CLR     @R2                 ;SET INTERRUPT ADDRESS
        MOVB    5(R1),@R2           ;...AS START ADDRESS+OFFSET
        ADD     R1,(R2)+
        CLR     @R2                 ;SET PRIORITY
        MOVB    6(R1),@R2
```

Figure 6-10
Position-independent Code

In both examples the program calling the device driver has correctly initialized its interrupt vector (VECTOR) within absolute memory locations Ø-377. The interrupt entry point offset is in byte DRIVER+5. (The contents of the driver table shows at DRIVER+5: ' .BYTE DVRINT,DRIVER.) The priority level is at byte DRIVER+6.

In the first example, the interrupt address is directly inserted into the absolute address of VECTOR. Neither of these addressing modes is position-independent.

The instruction to initialize the driver priority level uses an offset from the beginning of the driver code to the priority value, and places that value into the absolute address VECTOR+2, which is not position-independent. The final operation clearing the absolute address VECTOR+3 is also not position-independent.

In the position-independent code, operations are performed in registers wherever possible. The process of initializing registers is carefully planned to be position-independent. For example, the first two instructions obtain the starting address of the driver. The current PC value is loaded into R1, and the offset from the start of the driver to the current location is added to that value. Each of these operations is position-independent. The immediate mode value of VECTOR is loaded into R2, which places the absolute address of the transfer vector into a register for later use. The transfer vector is then cleared, and the offset for the driver starting address is loaded into the vector. The starting address of the driver is then added into the vector, giving the desired entry point to the driver. (This is equivalent to the first statement in Figure 6-9.) Since R2 has been updated to point to VECTOR+2, that location is then cleared and the priority level inserted into the appropriate byte.

The position-independent code demonstrates a principle of PDP-11 coding practice discussed earlier; that is, the programmer is advised to work primarily with register addressing modes wherever possible, relying on the setup mechanism to determine position-independence.

The MACRO Assembler provides the user with a way of checking the position-independence of the code. In an assembly listing, MACRO inserts a ' character following the contents of any word that requires the Linker to perform an operation. In some cases this character indicates a nonposition-independent instruction; in other cases it merely draws the user's attention to the use of a symbol that may or may not be position-independent. The cases that cause a ' character in the assembly listing follow.

1. Absolute mode symbolic references are flagged with an ' character when the reference is not position-independent. References are not flagged when they are position-independent (i.e., absolute). For example:

```
MOV  @#ADDR,R1          ;PIC ONLY IF ADDR IS ABSOLUTE.
```

2. Index mode and index deferred mode references are flagged with an ' character when the base is a symbolic label address (relocatable rather than an absolute value). For example:

```
MOV  ADDR(R1),R5        ;NON-PIC IF ADDR IS RELOCATABLE.
MOV  @ADDR(R1),R5       ;NON-PIC IF ADDR IS RELOCATABLE.
```

3. Relative mode and relative deferred mode are flagged with an ' character when the address specified is a global symbol. For example:

```
MOV  GLB1,R1            ;PIC WHEN GLB1 IS A GLOBAL SYMBOL.
MOV  @GLB1,R1           ;PIC WHEN GLB1 IS A GLOBAL SYMBOL.
```

   If the symbol is absolute, the reference is flagged and is not position-independent.

4. Immediate mode references to symbolic labels are always flagged with an ' character.

```
MOV  #3,R0              ;ALWAYS POSITION-INDEPENDENT.
MOV  #ADDR,R1           ;NON-PIC WHEN ADDR IS RELOCATABLE.
```

Examples of assembly listings containing the ' character are shown in Figure 6-11.

```
1 011744        ENDP2:                          ;END OF PASS 2
2                       .IF NDF  XCREF
3 011744 016702         MOV      CRFPNT,R2      ;ANY CREF IN PROGRESS?
         000142'
4 011750 001402         BFQ      8$             ;  NO
5 011752                CALL     CRFDMP         ;YES, DUMP AND CLOSE BUFFER
6 011756        8$:
7                       .ENDC
8 011756 005767         TST      BLKTYP         ;ANY OBJECT OUTPUT?
         000542'
9 011762 001423         BFQ      1$             ;  NO
10 11764                CALL     OPJDMP         ;YES, DUMP IT
11 11770 012767         MOV      #PLKT06,BLKTYP ;SET END
         000006
         000542'
12 11776                CALL     RLDDMP         ;DUMP IT
13                      .IF NDF  XFDABS
14 12002 032767         BTT      #FD.ABS,EDMASK ;ABS OUTPUT?
         000002
         000124'
15 12010 001010         BNE      1$             ;  NO
16 12012 016700         MOV      OBJPNT,R0
         000536'
17 12016 016720         MOV      ENDVEC+6,(R0)+ ;SET END VECTOR
         000044'
18 12022 010067         MOV      R0,OBJPNT
         000536'
19 12026                CALL     OBJDMP
20                      .FNDC
```

Figure 6-11
Assembly Listing Showing ' Character

Continued on next page

```
21 12Ø32 1Ø5767 1$:     TSTR    LLTPL+2          ;ANY LISTING OUTPUT?
         ØØØ546'
22 12Ø36 ØØ1474         BFQ     15$              ;  NO
23 12Ø4Ø Ø32767         BIT     #LC.SYM,LCMASK   ;SYMBOL TABLE SUPPRESSION?
         Ø4ØØØØ
         ØØØ11Ø'
```

Figure 6-11
Assembly Listing Showing ' Character