mac11 programming language

digital

DEC-15-LMCMA-A-D

MAC11 Programming Language

Order additional copies as directed on the Software Information page at the back of this document.

digital equipment corporation • maynard, massachusetts

First Printing, August 1974

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this manual.

The software described in this document is furnished to the purchaser under a license for use on a single computer system and can be copied (with inclusion of DIGITAL's copyright notice) only for use in such system, except as may otherwise be provided in writing by DIGITAL.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

Copyright © 1974 by Digital Equipment Corporation

The HOW TO OBTAIN SOFTWARE INFORMATION page, located at the back of this document, explains the various services available to DIGITAL software users.

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

The following are trademarks of Digital Equipment Corporation:

CDP	DIGITAL	INDAC	PS/8
COMPUTER LAB	DNC	KA10	QUICKPOINT
COMSYST	EDGRIN	LAB-8	RAD-8
COMTEX	EDUSYSTEM	LAB-8/e	RSTS
DDT	FLIP CHIP	LAB-K	RSX
DEC	FOCAL	OMNIBUS	RTM
DECCOMM	GLC-8	OS/8	RT-11
DECTAPE	IDAC	PDP	SABR
DIBOL	IDACS	PHA	TYPESET 8
			UNIBUS

CONTENTS

PART I

INTRODUCTION TO MAC11

PART II

DETAILS ON PROGRAMMING IN MAC11

PREFACE

This manual describes the PDP-11 MACRO-11 Assembler (MAC11) and Assembly Language and discusses briefly how to program the PDP-11 computer. It is recommended that the reader have with him copies of the PDP-11 Processor Handbook and, optionally, the PDP-11 Peripherals and Interfacing Handbook. References are made to these documents throughout this manual (although this document is complete, the additional material provides further details). The user is also advised to obtain a PDP-11 pocket Instruction List card for easy reference. (These items can be obtained from the Digital Software Distribution Center.)

This MACRO-11 Assembler operates under the PDP-15 DOS (Disk Operating System) Monitor in conjunction with PIREX, a multiprogramming executive running on a PDP-11 in the Unichannel 15 system.

Some notable features of MAC11 are:

1. Device and filename specifications for input
2. Error listing on command output device
3. Alphabetized, formatted symbol table listing
4. Conditional assembly directives
5. User defined macros
6. Extensive listing control

Associated Documents:

PDP-11/20 Processor Handbook 112.01071.1855

PDP-11 Peripherals and Interfacing Handbook 112.01071.1854

DOS-15 Users Manual, DEC-15-ODUMA-A-D

EDIT Utility Program, DEC-15-YWZB-DN6

PIP DOS Monitor Utility Program, DEC-15-UPIPA-B-D

The MAC11 assembler, a subset of the standard MACRO-11 assembler for the PDP-11, is specifically written for the Unichannel-15 system. Programs written for the MACRO-11 assembler will not necessarily assemble correctly with MAC11, and programs written for MAC11 will not necessarily assemble correctly with MACRO-11.

The MAC11 assembler generates only absolute binary output.

P A R T I

INTRODUCTION TO MAC11

CHAPTER 1

FUNDAMENTALS OF PROGRAMMING THE PDP-11

This Chapter presents some fundamental software concepts essential to efficient assembly language programming of the PDP-11 computer. A description of the hardware components of the PDP-11 family can be found in the two DEC paperback handbooks:

PDP-11 Processor Handbook (11/20 or 11/45 edition)

PDP-11 Peripherals and Interfacing Handbook

No attempt is made in this document to describe the PDP-11 hardware or the function of the various PDP-11 instructions. However, it is recommended that the reader become familiar with this material before proceeding.

The new PDP-11 programmer is advised to read this Chapter before reading further in this manual. The concepts in this Chapter will create a conceptual matrix within which explanations of the language fit. Since the techniques described herein work best with the PDP-11 and are used in PDP-11 system programs, they should be considered from the very start of your PDP-11 programming experience.

1.1 MODULAR PROGRAMMING

The PDP-11 family of computers lend themselves most easily to a modular system of programming. In such a system the programmer must envision the entire program and break it down into constituent subroutines. Modular development forces an awareness of the final system. Ideally, this should cause all components of the system to be considered from the very beginning of the development effort rather than patched into a partially-developed system. This provides for the best use of the PDP-11 hardware (as discussed later in this Chapter), and results in programs which are more easily modified than those coded with straight-line coding techniques.

To this end, flowcharting of the entire system is best performed prior to coding rather than during or after the coding effort. The programmer is then able to work on small portions of the program at a time. Subroutines of approximately one or two pages are considered desirable.

Modular programming practices maximize the usefulness of an installation's resources. Programmed modules can be used in other programs or systems having similar or identical functions without the expense of redundant development. Also software modules developed as functional entities are more likely to be free of serious logical errors as a result of the original programming effort. The use of such modules will simplify the development of later systems by incorporating proven pieces.

Modular development provides for ease of use and modification rather than simplifying the original development. While care must be taken in the beginning to ensure correct modular system development, the benefits of standardization to the generation of maintenance programmers which deal with a given assembly are many. (See also the notes under Commenting Assembly Language Programs.)

PDP-11 assembly language programming best follows a tree-like structure with the top of the tree being the final results and the base being the smallest component function. (The Assembler itself is a tree structure and is briefly described in Figure 1-1.)

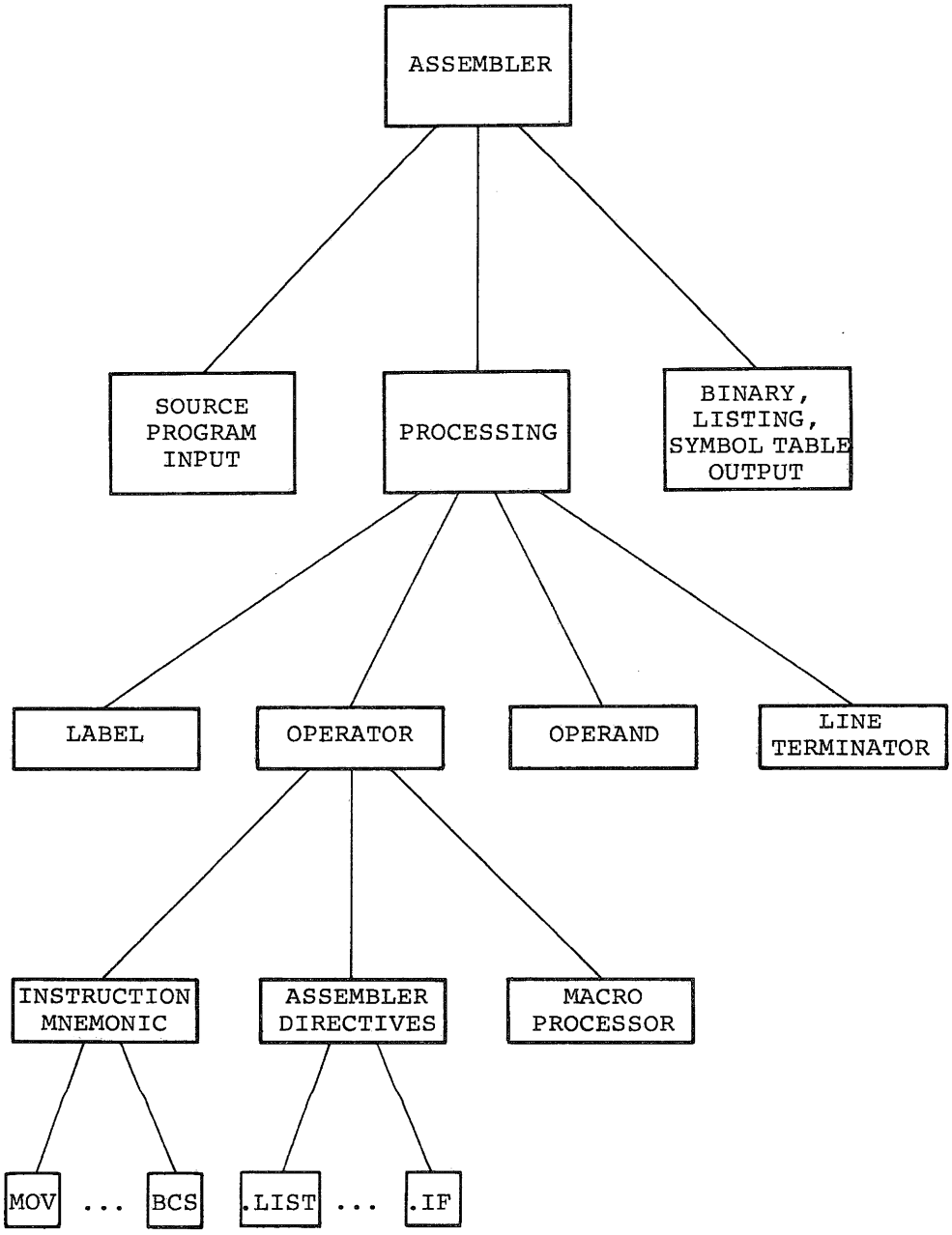


Figure 1-1
Problem Oriented Tree-Structure

1.1.1 Commenting PDP-11 Assembly Language Programs

When programming in a modular fashion, it is desirable to heavily comment the beginning of each subroutine, telling what that routine does: its inputs, outputs, and register usage.

Since subroutines are short and encompass only one operation it is not necessary to tell how the subroutine functions, but only what it does. An explanation of how a subroutine functions should be documented only when the procedure is not obvious to the reader. This enables any later inspection of an unclear subroutine to disclose the maximum amount of useful information to the reader.

1.1.2 Localized Register Usage

A useful technique in writing subroutines is to save all registers upon entering a subroutine and restore them prior to leaving the subroutine. This allows the programmer unrestricted use of the PDP-11 registers, including the program stack, during a subroutine.

Use of registers avoids 2- and 3-word addressing instructions. The code in Figure 1-2 compares the use of registers with symbolic addressing. Register use is faster and requires less storage space than symbolic addressing.

```
1                               .TFT
2    002060             10$:    CALL   20$              ;MOVE A CHARACTER
3    002064 003375              BGT    10$              ;LOOP IF GT ZERO
4    002066 001432              BFQ    19$              ;END IF ZERO
5    002070 114200              MOVB   -(R2),R0         ;TERMINATOR, BACK UP
                                                       ;POINTER
6    002072 020027              CMP    R0,#MT,MAX       ;END OF TYPE?
            177603
7    002076 101453              BLOS   22$              ;  YES
8    002100 010146              MOV    R1,-(SP)         ;REMEMBER READ POINTER
9    002102 016701              MOV    MSBARG,R1
            002034'
10    02106 005721              TST    (R1)+
11    02110 010203              MOV    R2,R3            ;  AND WRITE POINTER
12    02112 005400              NFG    R0               ;ASSUME MACRO
13    02114 026727              CMP    MSBTYP,#MT,MAC   ;TRUE?
            002026'
            177603
14    02122 001402              BFQ    12$              ;  YES, USE IT
15    02124 016700              MOV    MSBCNT,R0        ;GET ARG NUMBER
            002036'
16    02130 010302 12$:         MOV    R3,R2            ;RESET WRITE POINTER
17    02132        13$:         CALI   20$              ;MOVE A BYTE
18    02136 003375              BGT    13$              ;LOOP IF PNZ
19    02140 002402              BLT    14$              ;END IF LESS THAN ZERO
20    02142 005300              DFC    R0               ;ARE WE THERE YET?
21    02144 003371              BGT    12$              ;  NO
22    02146 105742 14$:         TSTB   -(R2)            ;YES, BACK UP POINTER
23    02150 012601              MOV    (SP)+,R1         ;RESET READ POINTER
24    02152 000742              BR     10$              ;END OF ARGUMENT
                                                       ; SUBSTITUTION
25
26    02154 010167 19$:         MOV    R1,MSBMRP        ;END OF LINE, SAVE
```

```
            002042'                                 ;POINTER
27    02160 052767        BIS     #LC,ME,LCFLAG     ;FLAG AS MACRO
            000400                                  ;EXPANSION
            000010'
28    02166 000726        BR      9$
29
30    02170 032701  20$:  BTT     #PPMB-1,R1        ;MACRO, END OF BLOCK?
            000017
31    02174 001003        BNE     21$               ;  NO
32    02176 016101        MOV     -PPMB(R1),R1      ;YES, POINT TO NEXT
            177760                                  ;BLOCK
33    02202 005721        TST     (R1)+             ;MOVE FAST LINK
34    02204 020227  21$:  CMP     R2,#LINBUF+SRCLFN     ;OVERFLOW?
35    02210 101404        BLOS    23$               ;  NO
36    02212               ERROR L                   ;YES, FLAG ERROR
37    02220 105742        TSTB    -(R2)             ;  AND MOVE POINTER
                                                    ;BACK
38    02222 112122  23$:  MOVB    (R1)+,(R2)+       ;MOVE CHAR INTO LINE
                                                    ;BUFFER
39    02224               RETURN
40
41    02226         22$:  CALL    ENDMAC            ;CLOSE MACRO
42    02232 000167        JMP     1$
            177326
43                        .ENDC
44
45
```

Figure 1-2
Segment of PDP-11 Code
Showing 1, 2, and 3-Word Instructions

1.1.3 Conditional Assemblies

Conditional assemblies are valuable in macro definitions. The expansion of a macro can be altered during assembly as a result of specific arguments passed and their use in conditionals. For example, a macro can be written to handle a given data item differently, depending upon the value of the item. Only a single algorithm need be expanded with each macro call. (Conditionals are described in detail in Section 5.7.)

Conditional assemblies can also be used to generate different versions of a program from a single source. This is usually done as a result of one or more symbols being either defined or undefined. Conditional assemblies are preferred to the creation of a multiplicity of sources. This principle is followed in the creation of PDP-11 system programs for the following reasons:

1. Maintenance of a single source program is easier, and guarantees that a change in one version of the program, which may affect other versions, is reflected automatically in all possible versions.

2. Distribution of a single source program allows a customer or individual user to tailor a system to his configuration and needs. and continue to update the system as the hardware environment or programming requirements change.

3. As in the case of maintenance, the debugging and checkout phase of a single program (even one containing many separate modules) is easier than testing several distinct versions of the same basic program.

1.2 REENTRANT CODE

Both the interrupt handling hardware and the subroutine call instructions (JSR, RTS, EMT, and TRAP) facilitate writing reentrant code for the PDP-11. Reentrant code allows a single copy of a given subroutine or program to be shared by more than one process or task. This reduces the amount of core needed for multi-task applications such as the concurrent servicing of peripheral devices.

On the PDP-11, reentrant code depends upon the stack for storage of temporary data values and the current processing status. Presence of information in the stack is not affected by the changing of operational control from one task to another. Control is always able to return to complete an operation which was begun earlier but not completed.

1.3 PREFERRED ADDRESSING MODES

Addressing modes are described in detail in Chapter 4. Basically, the PDP-11 programmer has eight types of register addressing and four types of addressing through the PC register. Those operations involving general register addressing take one word of core storage,

while symbolic addressing can use up to three words. For example:

```
MOV A,B             ;THREE WORDS OF STORAGE
MOV R0,R1           ;ONE WORD OF STORAGE
```

The user is advised to perform as many operations as possible with register addressing modes, and to use the remaining addressing modes to preset the registers for an operation. This technique saves space and time over the course of a program.

1.4 PARAMETER ASSIGNMENTS

Parameter assignments should be used to enable a program to be easily followed through the use of a symbolic cross reference (CREF listing). For example:

```
SYM=42
 .
 .
 .
MOV #SYM,R0
```

Another standard PDP-11 convention is to name the general registers as follows:

```
R0 = %0
R1 = %1
R2 = %2
R3 = %3
R4 = %4
R5 = %5
SP = %6      (processor stack pointer)
PC = %7      (program counter)
```

1.5 SPACE VS. TIMING TRADEOFFS

On the PDP-11 as on all computers, some techniques lead to savings in storage space and others lead to decreased execution time. Only the individual user can determine which is the best combination of the two for his application. It is the purpose of this section to describe several means of conserving core storage and/or saving time.

1.5.1 Trap Handler

The use of the trap handler and a dispatch table conserves core requirements in subroutine calling, but can lead to a decrease in execution speed due to indirect transfer of control. To illustrate, a subroutine call can be made in either of the following ways:

1. A JSR instruction which generally requires two PDP-11 words:

   ```
   JSR  R5,SUBA
   ```

   but is direct and fast.

2. A TRAP instruction which requires one in-line PDP-11 word:

```
TRAP N
```

but is indirect and slower. The TRAP handler must use N to index through a dispatch table of subroutine addresses and then JMP to the Nth subroutine in the table.

1.5.2 Register Increment

The operation:

```
CMPB  (R0)+,(R0)+
```

is preferable to:

```
TST  (R0)+
```

to increment R0 by 2, especially where the initial contents of R0 may be odd, but slower.

1.6 CONDITIONAL BRANCH INSTRUCTIONS

When using the PDP-11 conditional branch instructions, it is imperative that the correct choice be made between the signed and the unsigned branches.

SIGNED	UNSIGNED
BGE	BHIS (BCC)
BLT	BLO
BGT	BHI
BLE	BLOS (BCS)

A common error is to use a signed branch (e.g., BGT) when comparing two memory addresses. A problem occurs when the two addresses have opposite signs; that is, one address goes across the 16K (100000(8)) boundary. This type of coding error usually appears as a result of relinking at different addresses and/or a change in the size of the program.

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. Each line is terminated by either a line feed or a vertical-tab character (which increments the line count by 1) or a form-feed character (which increments both the line count and page count by 1).

Since the MAC11 Interface automatically appends a line feed at the end of every logical input line, the user need not concern himself with the statement terminator. However, a carriage return character not followed by a statement terminator generates an error flag. A legal statement terminator not immediately preceded by a carriage return causes the Assembler to insert a carriage return character for listing purposes.

An assembly language line can contain up to 80(10) characters (exclusive of the statement terminator). Beyond this limit, excess characters are ignored and generate an error flag.

2.1 STATEMENT FORMAT

A statement can contain up to four fields which are identified by order of appearance and by specified terminating characters. The general format of a MAC11 assembly language statement is:

```
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.

The Assembler interprets and processes these statements one by one, generating one or more binary instructions or data words or performing an assembly process. A statement must contain one of these fields and may contain all four types. (Blank lines are legal.)

Some statements have one operand, for example:

```
CLR  R0
```

while others have two,

```
MOV  #ERR,R2
```

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

MAC11 source statements are formatted with the DOS-15 EDIT program such that use of the TAB character causes the statement fields to be aligned. For example:

```
 Label     Operator    Operand          Comment
 Field      Field       Field            Field


MASK=-10
REGEXP:                                ;REGISTER EXPRESSION
          ABSEXP                       ;MUST BE ABSOLUTE
REGTST:   BIT         #MASK,VALUE      ;3 BITS?
          BEQ         REGERX           ;YES, OK
REGERR:   ERROR       R                ;NO, ERROR
REGERX:   MOV         #DEFFLG:REGFLG,MODE
          BIC         #MASK,VALUE
          BR          ABSERX
```

2.1.1 Label Field

A label is a user-defined symbol which is assigned the value of the current location counter and entered into the user-defined symbol table. The value of the label is absolute.

A label is a symbolic means of 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 100(8), the statement:

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

assigns the value 100(8) to the label ABCD. Subsequent reference to ABCD references location 100(8).

More than one label may appear within a single label field; each label within the field has the same value. For example, if the current location counter is 100(8), the multiple labels in the statement:

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

cause each of the three labels ABC, $DD, and A7.7 to be equated to the value 100(8).

The first six characters of a label are significant. An error code is generated if more than one label 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 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 one or more operands and/or a comment. Leading and trailing spaces and tabs 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) which 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 non-alphanumeric character (symbol component).

Consider the following examples:

```
MOV A,B    (space terminates the operator MOV)

MOV@A,B    (@ terminates the operator MOV)
```

When the statement line does not contain an operand or comment, the operator is terminated by a carriage return followed by a line feed, vertical tab or form feed character.

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

2.1.3 Operand Field

An operand is that part of a statement which is manipulated by the operator. Operands may be expressions, numbers, or symbolic 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 3.1.1). An operand may be preceded by an operator, label or other operand and followed by 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 space 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

usage, are ignored by the Assembler when appearing in the comment field.

The comment field may be preceded by one, any, none or all of the other three field types. Comments must begin with the semicolon character and end with a statement terminator.

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. A statement can be written:

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

or, using formatting characters, it can be written:

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

which is easier to read in the context of a source program listing.

Vertical formatting, i.e., page size, is controlled by the form feed character. A page of n lines is created by inserting a form feed (type the CTRL/FORM keys on the keyboard) after the nth line.

P A R T II

DETAILS ON PROGRAMMING IN MAC11

CHAPTER 3

SYMBOLS AND EXPRESSIONS

This Chapter describes the various components of legal MAC11 expressions: the Assembler character set, symbol construction, numbers, operators, terms and expressions.

3.1 CHARACTER SET

The following characters are legal in MAC11 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 0 through 9.

3. The characters . (period or dot) and $ (dollar sign),

The special characters are as follows:

Character	Designation	Function
carriage return		formatting character
line feed		
form feed		source statement terminators
vertical tab		
:	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
;	semi-colon	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 autoincrement indicator
-	minus sign	arithmetic subtraction operator or autodecrement 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	universal 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 3-1 and 3-2.

Table 3-1
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 3.8).
,	comma	A comma is a legal separator for both expressions and the argument operands.

Table 3-2
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 ↑\, where \ is any printing character, can be considered unary operators which 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 >, 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:

```
<A<B>C>
```

which reduces to:

```
A<B>C
```

and is considered to be one argument in both forms.

3.1.2 Illegal Characters

A character can be illegal in one of two ways:

1. A character which is not recognized as an element of the MAC11 character set is always an illegal character and causes immediate termination of the current line at that point, plus the output of an error flag in the assembly listing. For example:

```
LABEL←*A:     MOV  A,B
```

   Since the backarrow is not a recognized character, the entire line is treated as a:

```
.WORD   LABEL
```

   statement and is flagged in the listing.

2. A legal MAC11 character may be illegal in context. Such a character generates a Q error on the assembly listing.

3.1.3 Operator Characters

Legal unary operators under MAC11 are as follows:

Unary Operator	Explanation		Example
+	plus sign	+A	(positive value of A, equivalent to A)
-	minus sign	-A	(negative, 2's complement, value of A)
↑	up arrow, universal unary operator (this usage is described in greater detail in Sections 5.4.2 and 5.6.2).	↑C24(8)	(interprets the 1's complement value of 24(8))
		↑D127	(interprets 127 as a decimal number)
		↑O34	(interprets 34 as an octal number)
		↑B11000111	(interprets 11000111 as a binary value)

The unary operators as described above can be used adjacent to each other in a term. For example:

```
-%5
↑C↑O12
```

Legal binary operators under MAC11 are as follows:

Binary Operator	Explanation	Example	
+	addition	A+B	
-	subtraction	A-B	
*	multiplication	A*B	(16-bit product returned)
/	division	A/B	(16-bit quotient returned)
&	logical AND	A&B	
!	logical inclusive OR	A!B	

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. For example:

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

3.2 MAC11 SYMBOLS

There are three types of symbols: permanent, user-defined and macro. MAC11 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. The UST and MST are constructed as the source program is assembled; user-defined symbols are added to the table as they are encountered.

3.2.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics (Appendix B.3) and assembler directives (Chapters 5 and 6, Appendix B). These symbols are a permanent part of the Assembler and need not be defined before being used in the source program.

3.2.2 User-Defined and MACRO Symbols

User-defined symbols are those used as labels (Section 2.1.1) or defined by direct assignment (Section 3.3). 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 6.1). 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 following rules apply to the creation of user-defined and macro symbols:

1. The first character must not be a number.
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 following order:

1. Macro Symbol Table
2. Permanent Symbol Table
3. User-defined Symbol Table

A symbol found in the operand field is sought in the

1. User-defined Symbol Table
2. Permanent Symbol Table

in that order. The Assembler never expects to find a macro name in an operand field.

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.

All user-defined symbols are internal and must be defined within the current assembly.

3.3 DIRECT ASSIGNMENTS

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 and the specified value is associated with it. 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 is:

```
symbol = expression
```

Symbols take on the absolute attribute of their defining expression. For example:

```
        A = 1                   ;THE SYMBOL A IS EQUATED TO THE
                                ;VALUE 1.

        B='A-1&MASKLOW          ;THE SYMBOL B IS EQUATED TO THE
                                ;VALUE OF THE EXPRESSION.

C:      D = 3                   ;THE SYMBOL D IS EQUATED TO 3.

E:      MOV  #1,ABLE            ;LABELS C AND E ARE EQUATED TO THE
                                ;LOCATION OF THE MOV COMMAND.
```

The following conventions apply to direct assignment statements:

1. An equal sign (=) must separate the symbol from the expression defining the symbol value.

2. A direct assignment statement is usually placed in the operator field and may be preceded by a label and followed by a comment.

3. Only one symbol can be defined by any one 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
```

X and Y are both undefined throughout pass 1. X is undefined throughout pass 2 and causes a U error flag in the assembly listing.

3.4 REGISTER SYMBOLS

The eight general registers of the PDP-11 are numbered 0 through 7 and can be expressed in the source program as:

```
%0
%1

 .
 .
 .

%7
```

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

It is recommended that the programmer create and use symbolic names for all register references. A register symbol is 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:

```
8
9           000000    R0=%0      ;REGISTER DEFINITION
10          000001    R1=%1
11          000002    R2=%2
12          000003    R3=%3
13          000004    R4=%4
14          000005    R5=%5
15          000006    R6=%6
16          000006    SP=%6
17          000007    PC=%7
18          000007    R7=%7
19
```

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 fairly mnemonic, it is suggested the user follow this convention. Registers 6 and 7 are given special names because of their special functions, while registers 0 through 5 are given similar names to denote their status as general purpose registers.

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

The % character can be used with any term or expression to specify a register. (A register expression less than 0 or greater than 7 is flagged with an R error code.) For example:

```
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.

In certain cases a register can be referenced without the use of a register symbol or register expression; these cases are recognized through the context of the statement. An example is shown below:

```
JSR 5,SUBR    ;FIRST OPERAND FIELD MUST ALWAYS BE A REGISTER
```

3.5 LOCAL SYMBOLS

Local symbols are specially formatted symbols used as labels within a given range. Use of local symbols can achieve considerable savings in core space within the user symbol table. Core cost is one word for each local symbol in each local symbol block, as compared with four words of storage for each label stored in the user symbol table.

Local symbols provide a convenient means of generating labels for branch instructions, etc. Use of local symbols reduces the possibility of multiply-defined symbols within a user program and separates entry point symbols from local references. Local symbols are not referenced from outside their local symbol block.

Local symbols are of the form n$ where n is a decimal integer from 1 to 127, inclusive, and can only be used on word boundaries. Local symbols include:

```
 1$
27$
59$
104$
```

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 6.3.5 for further details). When using local symbols, the user is advised to first use the range from 1$ to 63$.

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 symbol labels. (Note that a statement of the form

   LABEL=.

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

2. The range of a single local symbol block can be delimited with the .ENABL LSB and the first symbolic label. The default for LSB is off.

For examples of local symbols and local symbol blocks, see Figure 3-3.

```
Ln.      Octal
No.      Expansion      Source Code              Comments

1                .SBTTL SECTOR INITIALIZATION
2
3
8
9
10               000000
11       00000           XCTPRG:
12       00000   012700          MOV     #IMPURE,R0
                 000000
13       00004   005020  1$:     CLR     (R0)+       ;CLEAR IMPURE AREA
14       00006   022700          CMP     #IMPTOP,R0
                 000040
15       00012   101374          BHI     1$
16
17               000000                              ;PASS     INITIALIZATION
                                                     ;CODE
18       00000           XCTPAS:
19       00000   012700          MOV     #IMPPAS,R0
                 000000
20       00004   005020  1$:     CLR     (R0)+       ;CLEAR IMPURE PART
21       00006   022700          CMP     #IMPTOP,R0
                 000040
22       00012   101374          BHI     1$
23
24               000000                              ;LINE     INITIALIZATION
                                                     ;CODE
25       00000           XCTLIN:
26       00020   012700          MOV     #IMPLIN,R0
                 000000
27       00004   005020  1$:     CLR     (R0)+
28       00006   022700          CMP     #IMPTOP,R0
                 000040
29       00012   101374          BHI     1$
30
```

```
31          000000                          ;MIXED MODE SECTOR
```

Figure 3-3
Assembly Source Listing of MAC11 Code
Showing Local Symbol Blocks

The maximum offset of a local symbol from the base of its local symbol blocks is 128 decimal words. Symbols beyond this range are flagged with an A error code.

3.6 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 #.,R0             ;. REFERS TO LOCATION A,
                                    ;I.E., THE ADDRESS OF THE
                                    ;MOV INSTRUCTION.
```

(# is explained in Section 5.9.)

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

```
        .=expression
```

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

Examples:

```
    .=500                           ;SET LOCATION COUNTER TO ABSOLUTE
                                    ;500

FIRST:    MOV   .+10,COUNT          ;THE LABEL FIRST HAS THE VALUE 500(8)
                                    ;.+10 EQUALS 510(8). THE CONTENTS OF
                                    ;THE LOCATION 510(8) WILL BE DEPOSITED
                                    ;IN LOCATION COUNT.

    .=520                           ;THE ASSEMBLY LOCATION COUNTER NOW
                                    ;HAS A VALUE OF ABSOLUTE 520(8).

SECOND:   MOV   .,INDEX             ;THE LABEL SECOND HAS THE VALUE 520(8)
                                    ;THE CONTENTS OF LOCATION 520(8), THAT
                                    ;IS, THE BINARY CODE FOR THE INSTRUCTION
                                    ;ITSELF, WILL BE DEPOSITED IN LOCATION
                                    ;INDEX.

    .=.+20                          ;SET LOCATION COUNTER TO ABSOLUTE 540 OF
                                    ;THE PROGRAM SECTION.
```

```
THIRD:    .WORD 0          ;THE LABEL THIRD HAS THE VALUE OF
                           ;ABSOLUTE 540.
```

Storage area may be reserved by advancing the location counter. For example, if the current value of the location counter is 1000, the direct assignment statement

```
.=.+100
```

reserves 100(8) bytes of storage space in the program. The next instruction is stored at 1100.

3.7 NUMBERS

The MAC11 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 5.4.1) or individual numbers can be treated as being of decimal, binary, or octal radix (see Section 5.4.2).

Octal numbers consist of the digits 0 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 2's complement form). Positive numbers may be preceded by a plus sign, although this is not required.

A number which is too large to fit into 16 bits (177777<n) is truncated from the left and flagged with a T error code in the assembly listing.

3.8 TERMS

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

1. A number, as defined in Section 3.7, whose 16-bit value is used.

2. A symbol, as defined earlier in the chapter. 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 whose basic value is used and whose arguments (if any) are ignored.

   c. an undefined symbol is assigned a value of zero and inserted in the user-defined symbol table.

3. An ASCII conversion using either an apostrophe followed by a single ASCII character or a double quote followed by two ASCII characters which results in a word containing the 7-bit ASCII value of the character(s). (This construction is explained in greater detail in Section 5.3.3.)

4. A term may also be 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).

3.9 EXPRESSIONS

Expressions are combinations of terms joined together by binary operators and which reduce to a 16-bit value. The operands of a .BYTE directive (see Section 5.3.1) are evaluated as word expressions before truncation to the low-order eight bits. Prior to truncation, the high-order byte must be zero or all ones (when byte value is negative, the sign bit is propagated).

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 is interpreted as +. A Q error flag is generated for each missing term or operator. For example:

```
TAG  !  LA 177777
```

is evaluated as

```
TAG  !  LA+177777
```

with a Q error flag on the assembly listing line.

The value of an expression is the value of the absolute part of the expression; e.g.,

```
        A = 5
        . = 20
TAG  :  MOV TAG+A,R0 ;SET R0 TO 25 (8).
```

CHAPTER 4

ADDRESSING MODES

The program counter (PC, register 7 of the eight general registers) always contains the address of the next word to be fetched; i.e., the address of the next instruction to be executed, or the second or third word of the current instruction.

In order to understand how the address modes operate and how they assemble, the action of the program counter must be understood. The key rule is:

> Whenever the processor implicitly uses the program counter to fetch a word from memory, the program counter is automatically incremented by two after the fetch.

That is, when an instruction is fetched, the PC is incremented by two, so that it is pointing to the next word in memory; and, if an instruction uses indexing (Sections 4.7, 4.8 and 4.11) the processor uses the program counter to fetch the base from memory. Hence, using the rule above, the PC increments by two, and now points to the next word.

The following conventions are used in this Section:

1. Let E be any expression as defined in Chapter 3.

2. Let R be a register expression. This is any expression containing a term preceded by a % character or a symbol previously equated to such a term.

   Examples:

```
R0 = %0     ;GENERAL REGISTER 0
R1 = R0+1   ;GENERAL REGISTER 1
R2 = 1+%1   ;GENERAL REGISTER 2
```

3. Let ER be a register expression or an expression in the range 0 to 7 inclusive.

4. Let A be a general address specification which produces a 6-bit mode address field as described in Sections 3.1 and 3.2 of the PDP-11 Processor Handbook (both 11/20 and 11/45 versions).

The addressing specifications, A, can be explained in terms of E, R, and ER as defined above. Each is illustrated with the single operand instruction CLR or double operand instruction MOV.

4.1 REGISTER MODE

The register contains the operand.

Format for A: R

```
Examples:       R0=%0           ;DEFINE R0 AS REGISTER 0
                CLR R0          ;CLEAR REGISTER 0
```

4.2 REGISTER DEFERRED MODE

The register contains the address of the operand.

Format for A: @R or (ER)

```
Examples:       CLR @R1         ;BOTH INSTRUCTIONS CLEAR
                CLR (1)         ;THE WORD AT THE ADDRESS
                                ;CONTAINED IN REGISTER 1
```

4.3 AUTOINCREMENT MODE

The contents of the register are incremented immediately after being used as the address of the operand. (See note below.)

Format for A: (ER)+

```
Examples:       CLR (R0)+       ;EACH INSTRUCTION CLEARS
                CLR (R0+3)+     ;THE WORD AT THE ADDRESS
                CLR (2)+        ;CONTAINED IN THE
                                ;SPECIFIED REGISTER AND
                                ;INCREMENTS THAT
                                ;REGISTER'S CONTENTS BY
                                ;TWO.
```

NOTE

Both JMP and ISR instructions using non-deferred autoincrement mode, autoincrement the register before its use on the PDP-11/20 (but not on the PDP-11/45 or 11/05). In double operand instructions of the addressing form %R,(R)+ or %R,-(R) where the source and destination registers are the same, the source operand is evaluated as the autoincremented or autodecremented value; but the destination register, at the time it is used, still contains the originally intended effective address.

In the following two examples, as executed on the PDP-11/20, R0 originally contains 100.

```
MOV   R0,(0)+    ;THE QUANTITY 102 IS
                 ;MOVED TO LOCATION 100

MOV   R0,-(0)    ;THE QUANTITY 76 IS MOVED
                 ;TO LOCATION 76
```

The use of these forms should be avoided as they are not compatible with the PDP-11/05 and 11/45.

A Z error code is printed with each instruction which is not compatible among all members of the PDP-11 family. This is merely a warning code.

4.4 AUTOINCREMENT DEFERRED MODE

The register contains the pointer to the address of the operand. The contents of the register are incremented after being used.

Format for A: @(ER)+

```
Example:    CLR @(3)+        ;CONTENTS OF REGISTER 3
                             ;POINT TO ADDRESS OF WORD
                             ;TO BE CLEARED BEFORE
                             ;BEING INCREMENTED BY TWO
```

4.5 AUTODECREMENT MODE

The contents of the register are decremented before being used as the address of the operand (see note under autoincrement mode).

Format for A: -(ER)

```
Examples:   CLR -(R0)        ;DECREMENT CONTENTS OF
            CLR -(R0+3)      ;REGISTERS 0, 3, AND 2 BY
            CLR -(2)         ;TWO BEFORE USING AS
                             ;ADDRESSES OF WORDS TO BE
                             ;CLEARED.
```

4.6 AUTODECREMENT DEFERRED MODE

The contents of the register are decremented before being used as the pointer to the address of the operand.

Format for A: @-(ER)

```
Example:    CLR @-(2)        ;DECREMENT CONTENTS OF
                             ;REGISTER 2 BY TWO BEFORE
                             ;USING AS POINTER TO
                             ;ADDRESS OF WORD TO BE
                             ;CLEARED.
```

4.7 INDEX MODE

The value of an expression E is stored as the second or third word of the instruction. The effective address is calculated as the value of E plus the contents of register ER. The value E is called the base.

```
Format for A:   E(ER)

Examples:       CLR X+2(R1)         ;EFFECTIVE ADDRESS IS X+2
                                    ;PLUS THE CONTENTS OF
                                    ;REGISTER 1.
                CLR -2(3)           ;EFFECTIVE
                                    ;ADDRESS IS -2 PLUS THE
                                    ;CONTENTS OF REGISTER 3.
```

4.8 INDEX DEFERRED MODE

An expression plus the contents of a register gives the pointer to the address of the operand.

```
Format for A:   @E(ER)

Example:        CLR @14(4)          ;IF REGISTER 4 HOLDS 100
                                    ;AND LOCATION 114 HOLDS
                                    ;2000, LOCATION 2000 IS
                                    ;CLEARED.
```

4.9 IMMEDIATE MODE

The immediate mode allows the operand itself to be stored as the second or third word of the instruction. It is assembled as an autoincrement of register 7, the PC.

```
Format for A:   #E

Examples:       MOV #100, R0        ;MOVE AN OCTAL 100 TO
                MOV #X, R0          ;REGISTER 0. MOVE THE
                                    ;VALUE OF SYMBOL X TO
                                    ;REGISTER 0.
```

The operation of this mode is explained as follows.

The statement MOV #100,R3 assembles as two words. These are:

```
0 1 2 7 0 3
0 0 0 1 0 0
```

Just before this instruction is fetched and executed, the PC points to the first word of the instruction. The processor fetches the first word and increments the PC by two. The source operand mode is 27 (autoincrement the PC). Thus the PC is used as a pointer to fetch the operand (the second word of the instruction) before being incremented by two, to point to the next instruction.

4.10 ABSOLUTE MODE

Absolute mode is the equivalent of immediate mode deferred. @#E specifies an absolute address which is stored in the second or third word of the instruction. Absolute mode is assembled as an autoincrement deferred of register 7, the PC.

```
Format for A:   @#E

Examples:       MOV @#100,R0        ;MOVE THE VALUE OF THE
                                    ;CONTENTS OF LOCATION 100
                CLR @#X             ;TO REGISTER 0. CLEAR
                                    ;THE CONTENTS OF THE
                                    ;LOCATION WHOSE ADDRESS
                                    ;IS X.
```

4.11 RELATIVE MODE

Relative mode is the normal mode for memory references.

```
Format for A:   E

Examples:       CLR 100             ;CLEAR LOCATION 100.
                MOV X,Y             ;MOVE CONTENTS OF
                                    ;LOCATION X TO LOCATION
                                    ;Y.
```

Relative mode is assembled as index mode, using register 7, the PC, as the index register. The base of the address calculation, which is stored in the second or third word of the instruction, is not the address of the operand (as in index mode), but the number which, when added to the PC, becomes the address of the operand. Thus, the base is X-PC, which is called an offset. The operation is explained as follows:

If the statement MOV 100,R3 is assembled at absolute location 20, the assembled code is:

```
Location 20:    0 1 6 7 0 3
Location 22:    0 0 0 0 5 4
```

The processor fetches the MOV instruction and adds two to the PC so that it points to location 22. The source operand mode is 67; that is, indexed by the PC. To pick up the base, the processor fetches the word pointed to by the PC and adds two to the PC. The PC now points to location 24. To calculate the address of the source operand, the base is added to the designated register. That is, BASE+PC=54+24=100, the operand address.

Since the Assembler considers "." as the address of the first word of the instruction, an equivalent index mode statement would be:

```
MOV 100-.-4(PC),R3
```

This mode is called relative because the operand address is calculated relative to the current PC. The base is the distance or offset (in bytes) between the operand and the current PC. If the operator and its operand are moved in memory so that the distance between the operator and data remains constant, the instruction will operate correctly anywhere in core.

4.12 RELATIVE DEFERRED MODE

Relative deferred mode is similar to relative mode, except that the expression, E, is used as the pointer to the address of the operand.

Format for A: @E

```
Example:        MOV @X,R0           ;MOVE THE CONTENTS OF THE
                                    ;LOCATION WHOSE ADDRESS IS
                                    ;IN X INTO REGISTER 0.
```

4.13 TABLE OF MODE FORMS AND CODES

Each instruction takes at least one word. Operands of the first six forms listed below do not increase the length of an instruction. Each operand in one of the other modes, however, increases the instruction length by one word.

Form	Mode	Meaning
R	0n	Register mode
@R or (ER)	1n	Register deferred mode
(ER)+	2n	Autoincrement mode
@(ER)+	3n	Autoincrement deferred mode
-(ER)	4n	Autodecrement mode
@-(ER)	5n	Autodecrement deferred mode

where n is the register number.

Any of the following forms adds one word to the instruction length:

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

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

NOTE

1. An alternate form for @R is (ER). However, the form @(ER) is equivalent to @0(ER).

2. 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 @#100 clears absolute location 100 even if the instruction is moved from the point at which it was assembled.

4.14 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 offset (7 bits plus sign) which specifies the branch address relative to the PC. The hardware calculates the branch address as follows:

1. Extend the sign of the offset through bits 8-15.

2. Multiply the result by 2. This creates a word offset rather than a byte offset.

3. Add the result to the PC to form the final branch address.

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

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

Since PC = .+2, we have

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

The EMT and TRAP instructions do not use the low-order byte of the word. This allows information to be transferred to the trap handlers in the 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(8)) it is truncated to eight bits and a T error flag is generated.

P A R T III

MAC11 ASSEMBLER DIRECTIVES

Chapters 5 and 6 describe all MAC11 directives. Directives are statements which cause the Assembler to perform certain processing operations. Chapter 5 describes several types of directives including those to control symbol interpretation, listing header material, program sections, data storage format, assembly listings, and floating-point formats. Chapter 6 describes those directives having to do with macros, macro arguments, and repetitive coding situations.

Assembler directives can be preceded by a label, subject to restrictions associated with specific directives, and followed by a comment. An assembler directive occupies the operator field of a MAC11 source line. Only one directive can be placed on any one line. Zero, one, or more operands can occupy the operand field; legal operands differ with each directive and may be symbols, expressions, or arguments.

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 MAC11 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 LINE 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 are as follows (these arguments can be used singly or in combination):

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 expressions.
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.

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 tables of contents on pass 1 of the assembly (see Section 5.1.4 describing the .SBTTL directive). The full assembly listing is printed during pass 1 of the assembly.
TTM	Teletype mode	Controls 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 Teletype mode is line printer mode, which is shown in Figure 6-1.
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 5-1. Notice that binary extensions for statements generating more than one word are spread horizontally on the source line.

```
 1 001766                                GETLIN:                         ;GET AN INPUT LINE
 2 001766                                       SAVREG
 3 001772  016700  000020'               1$:    MOV     FFCNT,R0        ;ANY RESERVED FF'S?
 4 001776  001420                               BEQ     31$             ;  NO
 5 002000  060067  000022'                      ADD     R0,PAGNUM       ;YES, UPDATE PAGE NUMBER
 6 002004  012767  177777  000026'              MOV     #-1,PAGEXT
 7 002012  005067  000012'                      CLR     LINNUM          ;INIT NEW CREF SEQUENCE
 8 002016  005067  000020'                      CLR     FFCNT
 9 002022  005067  000016'                      CLR     SEQEND
10 002026  005767  000000'                      TST     PASS
11 002032  001402                               BEQ     31$
12 002034  005067  000010'                      CLR     LPPCNT
13 002040  012702  001712'              31$     MOV     #LINBUF,R2
14 002044  010267  000012'                      MOV     R2,LCBEGL       ;SEAT UP BEGINNING
15 002050  012767  002116'  000014'            MOV     #LINEND,LCENDL  ;  AND END OF LINE MARKERS
17 002056  005767  000200'                      TST     SMLCNT          ;IN SYSTEM MACRO?
18 002062  001145                               BNE     40$             ;  YES, SPECIAL
21 002064  016701  002214'                      MOV     MSBMRP,R1       ;ASSUME MACRO IN PROGRESS
22 002070  001166                               BNE     10$             ;BRANCH IF SO
24 002072  012701  000756'                      MOV     #SRCBUF,R1
25 002076                                       .WAIT   #SRCLNK
26 002104  005267  000012'                      INC     LINNUM
27 002110  116700  000753'                      MOVB    SRCHDR+3,R0     ;GET CODE BYTE
28 002114  032700  000047                       BIT     #047,R0         ;ANYTHING BAD?
29 002120  001403                               BEQ     32$             ;  NO
30 002122                                       ERROR   L               ;YES, ERROR
31 002130  106100                       32$:    ROLB    R0              ;EOF?
32 002130  100014                               BPL     2$              ;  NO
33 002134  056767  000006'  000004'            BIS     CSISAV,ENDFLG
34 002142  001003                               BNE     34$
```

Figure 5-1
Example of MAC11 Line Printer Listing
(132-column line printer)

5.1.2 Page Headings

The MAC11 Assembler outputs each page in the format shown in Figure 5-1, Line Printer Listing. On the first line of each listing page the Assembler prints (from right to left):

1. title taken from .TITLE directive.
2. assembler version identification
3. 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 listing output. The name is the first symbol following the directive and must be six Radix-50 characters or less (any characters beyond the first six are ignored). Non-Radix 50 characters are not acceptable. For example:

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

causes the listing output of the assembled program to be named PROG (this name is distinguished from the filename of the binary output specified in the command string to the Assembler).

If there is no .TITLE statement, the default name assigned to the first listing output is

```
.MAIN.
```

The first tab or space following the .TITLE directive is not considered part of the listing output 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 listing output.

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, MAC11 automatically prints a table of contents for the listing containing the line sequence number and text of each .SBTTL directive in the program.

An example of the table of contents is shown in Figure 5-2. Note that the first word of the subtitle heading is not limited to six characters since it is not a module name.

```
MACRO   V1A                    MACRO V1A
                          TABLE OF CONTENTS


    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
   40-  1       ASSIGNMENT PROCESSOR
   41-  1       OP CODE PROCESSOR
   48-  1       EXPRESSION TO CODE-ROLL CONVERSIONS
   50-  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       KEPBOARD HANDLERS
   80-  1       OBJECT CODE HANDLERS
   88-  1       LISTING OUTPUT
   92-  1       I/O BUFFERS
   93-  1       EXPRESSION EVALUATOR
   99-  1       TERM EVALUATOR
  103-  1       SYMBOL/CHARACTER HANDLERS
  109-  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 directive.

Figure 5-2
Assembly Listing Table of Contents

5.1.5 Page Ejection

There are several means of obtaining a page eject in a MAC11 assembly listing:

1. After a line count of 58 lines, MAC11 automatically performs a page eject to skip over page perforations on line printer paper and to formulate terminal output into pages.

2. A form feed character used as a line terminator (or 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.

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

   ```
   .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.

5.2 FUNCTIONS: .ENABL AND .DSABL DIRECTIVES

Several functions are provided by MAC11 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 defined below.

The following table describes the symbolic arguments and their associated functions in the MAC11 language:

Symbolic Argument	Function
CDR	The statement .ENABL CDR causes source columns 73 and greater to be treated as comment. This accommodates sequence numbers in card columns 72-80.
LC	Enabling of this function causes the Assembler to accept lower case ASCII input instead of converting it to upper case.
LSB	Enable or disable a local symbol block. While a local symbol block is normally entered by encountering a new symbolic label, .ENABL LSB forces a local symbol block which is not terminated until a label following the

.DSABL LSB statement is encountered. The default case is .DSABL LSB.

PNC — The statement .DSABL PNC inhibits binary output until an .ENABL PNC is encountered. The default case is .ENABL PNC.

An incorrect argument causes the directive containing it to be flagged as an error.

5.3 DATA STORAGE DIRECTIVES

A wide range of data and data types can be generated with the following directives and assembly characters:

```
.BYTE
.WORD
'
"
.ASCII
.ASCIZ
.RAD50
↑B
↑D
↑O
```

These facilities are explained in the following Sections.

5.3.1 .BYTE

The .BYTE directive is used to generate successive bytes of data. The directive is of the form:

```
.BYTE exp                ;WHICH STORES THE OCTAL EQUIVALENT
                         ;OF THE EXPRESSION exp IN THE NEXT
                         ;BYTE.

.BYTE exp1,exp2,...      ;WHICH STORES THE OCTAL EQUIVALENTS
                         ;OF THE LIST OF EXPRESSIONS IN
                         ;SUCCESSIVE BYTES.
```

where 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
.=410
.BYTE  ↑D48,SAM          ;060 (OCTAL EQUIVALENT OF 48 DECIMAL)
                         ;IS STORED IN LOCATION 410, 005 IS
                         ;STORED IN LOCATION 411.
```

If the high order byte of the expression equates to a value other than 0 or -1, it is truncated to the low-order eight bits and flagged with a T error code.

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

```
.=420
.BYTE ,,                ;ZEROES ARE STORED IN BYTES 420,
                        421, AND 422.
```

5.3.2 .WORD

The .WORD directive is used to generate successive words of data. The directive is of the form:

```
.WORD exp               ;WHICH STORES THE OCTAL EQUIVALENT
                        ;OF THE EXPRESSION exp IN THE NEXT
                        ;WORD

.WORD expl,exp2,...     ;WHICH STORES THE OCTAL EQUIVALENTS
                        ;OF THE LIST OF EXPRESSIONS IN
                        ;SUCCESSIVE WORDS.
```

where a legal expression must result in sixteen 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=0
.=500
.WORD 177535,.+4,SAL    ;STORES 177535, 506, AND 0 IN
                        ;WORDS 500, 502, AND 504.
```

If an expression equates to a value of more than sixteen 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:

```
.=500
Word  ,5,               ;STORES 0, 5, AND 0 IN LOCATIONS 500
                        ;502, AND 504.
```

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. 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:

```
        .=440           ;THE OP-CODE FOR MOV, WHICH IS 010000,
LABEL:  +MOV,LABEL      ;IS STORED ON LOCATION 440.
                        ;440 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 which is not recognized as a macro call, an instruction mnemonic or 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
```

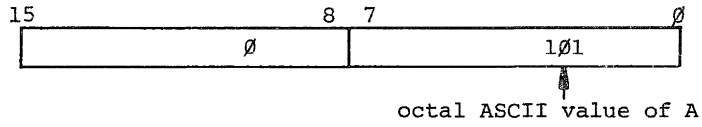
Two error codes result: Q occurs because an expression operator is missing between MOR and A, and a U occurs if MOR is undefined. Two words are then generated: one for MOR A and one for B.

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,R0
```

results in the following sixteen bits being moved into R0:



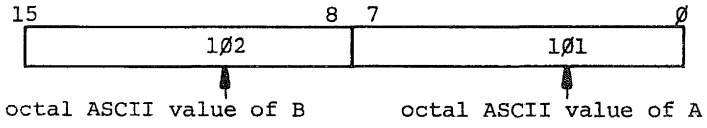
The ' character is never followed by a carriage return, null, rubout, line feed or form feed. (For another use of the ' character, see Section 5.3.6.)

```
STMNT:
        GETSYM
        BEQ     4$
        CMPB    @CHRPNT,#':     ;COLON DELIMITS LABEL FIELD.
        BEQ     LABEL
        CMPB    @CHRPNT,#'=     ;EQUAL DELIMITS
        BEQ     ASGMT           ;ASSIGNMENT PARAMETER.
```

A double quote followed by two characters results in a word in which the 7-bit ASCII representations of the two characters are placed. For example:

```
MOV  #"AB,R0
```

results in the following word being moved into R0:



The " character is never followed by a carriage return, null, rubout, line feed or form feed. For example:

```
;DEVICE NAME TABLE

DEVNAM:   .WORD     "DF       ;RF DISK
          .WORD     "DK       ;RK DISK
          .WORD     "DP       ;RP DISK
DEVNKB:   .WORD     "KB       ;TTY KEYBOARD
          .WORD     "DT       ;DECTAPE
          .WORD     "LP       ;LINE PRINTER
          .WORD     "PR       ;PAPER TAPE READER
          .WORD     "PP       ;PAPER TAPE PUNCH
          .WORD     "CR       ;CARD READER
          .WORD     "MT       ;MAGTAPE
          .WORD     0         ;TABLE'S END
```

5.3.4 .ASCII

The .ASCII directive translates character strings into their 7-bit .ASCII equivalents for use in the source program. The format of the .ASCII directive is as follows:

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

where:

character string	is a string of any acceptable printing ASCII characters. The string may not include null (blank) characters, rubout, carriage return, line feed, vertical tab, or form feed. 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 ; < and = characters and any character within the string.

As an example:

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

        .ASCII BC/<15><12>/DEF/     ;STORES A,B,C,15,12,D,E,F IN
                                    ;CONSECUTIVE BYTES.

        .ASCII  /<AB>/              ;STORES <,A,B,> IN 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:

Example	ASCII string Generated	Notes
.ASCII ;ABC;/DEF/	A B C D E F	Acceptable, but not a recommended procedure.
.ASCII /ABC/;DEF;	A B C	;DEF; is treated as a comment and ignored.
.ASCII /ABC/=DEF=	A B C D E F	Acceptable, but not recommended procedure.
.ASCII =DEF=		The assignment .ASCII=DEF is performed and a Q-error is 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:

```
        .
        .
        .
    MOV  #HELLO,R1
    MOV  #LINBUF,R2
X:  MOVB (R1)+,(R2)+
    BNE  X
        .
        .
        .
HELLO:  .ASCIZ  <CR><LF>/MAC11 V1A/<CR><LF>  ;INTRO MESSAGE
                        .
                        .
                        .
```

5.3.6 .RAD50

The .RAD50 directive allows the user the capability to handle symbols in Radix-50 coded form (this form is sometimes referred to as MOD40 and is used in PDP-11 system programs). Radix-50 form allows three characters to be packed into sixteen bits; therefore, any 6-character symbol can be held in two words. The form of the directive is:

```
.RAD50          /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, 0 through 9, dollar ($), dot (.) 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 carriage return, semicolon, or matching delimiter. For example:

```
.RAD50    /ABC            ;PACK ABC INTO ONE WORD.
.RAD50    /AB/            ;PACK AB (SPACE) INTO ONE WORD.
.RAD50    //              ;PACK 3 SPACES INTO ONE WORD.
.RAD50    /ABCD/          ;PACK ABC INTO FIRST WORD AND
                          ;D SPACE SPACE INTO SECOND WORD.
```

Each character is translated into its Radix-50 equivalent as indicated in the following table:

Character	Radix-50 Equivalent (octal)
(space)	0
A-Z	1-32
$	33
.	34
0-9	36-47

Note that another character could be defined for code 35, which is currently unused.

The Radix-50 equivalents for characters 1 through 3 (C1, C2, C3) are combined as follows:

```
Radix 50 value = ((C1*50)+C2)*50+C3
```

For example:

```
Radix-50 value of ABC is ((1*50)+2)*50+3 or 3223
```

See Appendix A for a table to quickly determine 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:

```
.ASCII <101>                    ;EQUIVALENT TO .ASCII/A/

.RAD50 /AB/<35>                 ;STORES 3255 IN NEXT WORD
```

```
CHR1=1
CHR2=2
CHR3=3
   .
   .
   .
.RAD50<CHR1><CHR2><CHR3>          ;EQUIVALENT TO .RAD50/ABC/
```

5.4 RADIX CONTROL

5.4.1 .RADIX

Numbers used in a MAC11 source program are initially considered to be octal numbers. However, the programmer has the option of declaring the following radices:

2, 4, 8, 10

This is done via the .RADIX directive, of the form:

```
.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 (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 nor rely on radix settings from the .RADIX directive. The temporary radix control characters should be used within a macro definition. (↑D, ↑O, and ↑B 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.

MACll has three unary operators to provide a single interpretation in a given radix within another radix as follows:

```
↑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  00001101
↑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.

PAL-11R contains a feature, which is maintained for compatibility in MACll, allowing a temporary radix change from octal to decimal by specifying a decimal radix number with a "decimal point". For example:

```
 100.     (144(8))
1376.     (2540(8))
 128.     (200(8))
```

5.5 LOCATION COUNTER CONTROL

The four directives which control movement of the location counter are .EVEN and .ODD which move the counter a maximum of one byte, and .BLKB and .BLKW which 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 a .EVEN directive are ignored.

The .EVEN directive is used as follows:

```
.ASCIZ /THIS IS A TEST/

.EVEN                   ;ASSURES NEXT STATEMENT

                        ;BEGINS ON A WORD BOUNDARY.

.WORD XYZ
```

5.5.2 .ODD

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

```
;CODE TO MOVE DATA FROM AN INPUT LINE
;TO A BUFFER

        N=5                     ;BUFFER HAS 5 WORDS
          .
          .
          .

        .ODD
        .BYTE   N*2             ;COUNT=2N BYTES
BUFF:   .BLKW   N               ;RESERVE BUFFER OF N WORDS

          .
          .
          .

        MOV     #BUFF,R2        ;ADDRESS OF EMPTY BUFFER IN R2
        MOV     #LINE,R1        ;ADDRESS OF INPUT LINE IS IN R1
        MOVB    1(R2),R0        ;GET COUNT STORED IN BUFF-1 IN R0
AGAIN:  MOVB    (R1)+,(R2)+     ;MOVE BYTE FROM LINE INTO BUFFER
        BEQ     DONE            ;WAS NULL CHARACTER SEEN?
        DEC     R0              ;DECREMENT COUNT
        BNE     AGAIN           ;NOT = 0, GET NEXT CHARACTER

          .
          .
          .

        CLRB    -(R2)           ;OUT OF ROOM IN BUFFER, CLEAR LAST
DONE:                           ;WORD

          .
          .
          .

LINE:   .ASCIZ                  /TEXT/
```

In this case, .ODD is used to place the buffer byte count in the byte preceding the buffer, as follows:

```
                COUNT                   BUFF-2

                                        BUFF
```

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. The two directives are of the form:

```
.BLKB  exp

.BLKW  exp
```

where:

exp is the number of bytes or words to reserve. If no argument is present, 1 is the assumed default value. Any legal expression which is completely defined at assembly time and produces an absolute number is legal.

For example:

```
1 000000          PASS:   .BLKW
2                                       ;NEXT GROUP MUST STAY TOGETHER
3 000002          SYMBOL: .BLKW  2      ;SYMBOL ACCUMULATOR
4 000006          MODE:
5 000006          FLAGS:  .BLKB  1      ;FLAG BITS
6 000007          SECTOR: .BLKB  1      ;SYMBOL/EXPRESSIONS TYPE
7 000010          VALUE:  .BLKW  1      ;EXPRESSION VALUE
8 00012          RELLVL: .BLKW  1
9                        .BLKW  2       ;END OF GROUPED DATA
10
11 00020          CLCNAM: .BLKW  2      ;CURRENT LOCATION COUNTER SYMBOL
12 00024          CLCFGS: .BLKB  1
13 00025          CLCSEC: .BLKB  1
14 00026          CLCLOC: .BLKW  1
15 00030          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 TERMINATING DIRECTIVES

5.6.1 .END

The .END directive indicates the physical end of the source program. The .END directive is of the form:

```
.END  exp
```

where:

exp is an optional argument which, if present, indicates the program entry point, i.e., the transfer address.

At the conclusion of the first assembly pass, upon encountering the END statement, MAC11 prints:

```
END OF PASS 1
```

and attempts to reread the source file(s) to perform pass 2. If the source file is on a disk, DECtape, or magtape device no further operator action is necessary. If the source file is on paper tape an IOPS 4 message is printed; the user is expected to reposition the tape in the reader and type ↑R (for CONTINUE).

5.7 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 extensively to allow several variations of a program to be generated from the source program.

The general form of a conditional block is as 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 below.
argument(s)	are a function of the condition to be tested.
range	is the body of code which is included in the assembly or ignored depending upon whether the condition was met.

The following are the allowable conditions:

Conditions

POSITIVE	COMPLEMENT	ARGUMENTS	ASSEMBLE BLOCK IF
EQ	NE	expression	expression=0 (or =0)
GT	LE	expression	expression>0 (or <0)
LT	GE	expression	expression<0 (or >0)
DF	NDF	symbolic argument	symbol is defined (or undefined)
B	NB	macro-type argument	argument is a blank (or not blank)

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

NOTE

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

```
<A,B,C>
↑/124/
```

For example:

```
.IF EQ    ALPHA+1          ;ASSEMBLE IF ALPHA+1=0
     .
     .
     .
.ENDC
```

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

```
& logical AND operator
! logical inclusive OR operator
```

For example:

```
.IF DF SYM1 & SYM2
     .
     .
     .
.ENDC
```

assembles if both SYM1 and SYM2 are defined.

5.7.1 Subconditionals

Subconditionals may be placed within conditional blocks to indicate:

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 non-contiguous 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.

There are three subconditional directives, as follows:

Subconditional	Function
.IFF	The code following thi s statement up to the next subconditional or end of the conditional block is included in the program providing 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 providing 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 the subconditionals is the value of the condition upon entering the conditional block. Subconditionals are used within outer level conditional blocks. Subconditionals are ignored within nested, unsatisfied condition blocks. For example:

```
.IF    DF    SYM            ;ASSEMBLE BLOCK IF SYM IS DEFINED
.IFF
  .                         ;ASSEMBLE THE FOLLOWING CODE ONLY IF
  .                         ;SYM IS UNDEFINED.
  .
  .IFT                      ;ASSEMBLE THE FOLLOWING CODE ONLY IF
  .                         ;SYM IS DEFINED.
  .
  .
  .IFTF                     ;ASSEMBLE THE FOLLOWING CODE
  .                         ;UNCONDITIONALLY.
  .
  .
.ENDC
```

```
.IF    DF    X              ;ASSEMBLY TESTS FALSE
.IF    DF    Y              ;TESTS FALSE
.IFF                        ;NESTED CONDITIONAL
  .                         ;IGNORED
  .
  .
.IFT                        ;NOT SEEN
  .
  .
  .
.ENDC
```

However,

```
.IF   DF   X                ;TESTS TRUE
.IF   DF   Y                ;TESTS FALSE
.IFF                        ;IS ASSEMBLED
  .
  .
  .
.IFT                        ;NOT ASSEMBLED
  .
  .
  .
.ENDC
```

5.7.2 Immediate Conditionals

An immediate conditional directive is a means of writing a 1-line conditional block. 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 Section 5.7.
arg	is the argument associated with the condition specified; that is, either an expression, symbol, or macro-type argument, as described in Section 5.7.
statement	is the statement to be executed 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    FPP,BEQ,ALPHA
```

or included as part of the conditional statement:

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

5.7.3 PAL-11R Conditional Assembly Directives

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

Directive	Arguments	Assemble Block if
.IFZ or .IFEQ	expression	expression=0
.IFNZ or .IFNE	expression	expression=0
.IFL or .IFLT	expression	expression<0
.IFG or .IFGT	expression	expression>0
.IFLE	expression	expression< or =0
.IFGE	expression	expression> or =0
.IFDF	logical expression	expression is true (defined)
.IFNDF	logical expression	expression is false (undefined)

The rules governing the usage of these directives are now the same as for the MACRO-11 conditional assembly directives previously described. Conditional assembly blocks must end with the .ENDC directive and are limited to a nesting depth of 16(10) levels (instead of the 127(10) levels allowed under PAL-11R).

CHAPTER 6

MACRO DIRECTIVES

6.1 MACRO DEFINITION

It is often convenient in assembly language programming to generate a recurring coding sequence with a single statement. In order to do this, the desired coding sequence is first defined with dummy arguments as a macro. Once a macro has been defined, a single statement calling the macro by name with a list of real arguments (replacing the corresponding dummy arguments in the definition) generates the correct sequence or expansion.

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	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 that 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.

An example of a macro definition is shown below:

```
.MACRO TYPMSG MESSGE  ;TYPE A MESSAGE
JSR R5,TYPMSG
.WORD MESSGE
.ENDM
```

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:

```
.MACRO  ALTR  N,A,B
  .
  .
  .
.IF     EQ,N        ;START CONDITIONAL BLOCK
  .
  .
  .
.MEXIT              ;EXIT FROM MACRO DURING CONDITIONAL BLOCK
.ENDC               ;END CONDITIONAL BLOCK
  .
  .
  .
.ENDM               ;NORMAL END OF MACRO
```

In an assembly where N=0, the .MEXIT directive terminates the macro expansirn.

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

6.1.4 MACRO Definition Formatting

A form feed character used as a line terminator on a MAC11 source statement (or 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.
real arguments	are those symbols, expressions, and values which 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:

```
ABS:    MOV @R0,R1          ;ABS IS USED AS A LABEL
          .
          .
          .
        BR ABS              ;ABS IS CONSIDERED A LABEL
          .
          .
          .
        ABS #4,ENT,LAR      ;CALL MACRO ABS WITH 3 ARGUMENTS
```

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

6.3 ARGUMENTS TO 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 3.1.1. For example:

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

Arguments which 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 follows:

```
REN  ↑/MOV X,Y/,#44,WEV
```

which 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 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. 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.

```
.MACRO  LEVEL1  DUM1,DUM2
LEVEL2  DUM1
LEVEL2  DUM2
.ENDM
```

```
.MACRO  LEVEL2  DUM3
DUM3
ADD #10,R0
MOV R0, (R1)+
.ENDM
```

A call to the LEVEL1 macro:

```
LEVEL1  <<MOV X,R0>>,<<CLR R0>>
```

causes the following expansion:

```
MOV X,R0
ADD #10,R0
MOV R0, (R1)+
CLR R0
ADD #10,R0
MOV R0,(R1)+
```

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 Special Characters

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

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

generates the following code:

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

6.3.3 Numeric Arguments Passed as Symbols

When passing macro arguments, a useful capability is to pass a symbol which 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 functions are defined in context. For example:

```
        B=0
        .MACRO  INC A,B
        CNT     A,\B
        B=B+1
        .ENDM
        .MACRO CNT  A,B
A'B:    .WORD         /SEE SEC.6.3.6 FOR EXPLANATION OF 'B.
        .ENDM
          .
          .
          .
        INC  X,C
```

The macro call would expand to:

```
        X0:  .WORD
```

A subsequent identical call to the same macro would generate:

```
        X1:  .WORD
```

and so on for later calls. The two macros are necessary because the dummy value of B cannot be updated in the CNT macro. In the CNT macro, the number passed is treated as a string argument. (Where the value of the real argument is 0, a single 0 character is passed to the macro expansion.)

The number being passed can also be used to 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
.ASCII  /SYM/             ;ON A UNIQUE TWO DIGIT VALUE FOR
.ENDM                     ;EACH POSSIBLE CONDITIONAL ASSEMBLY
.MACRO  OUT  ARG          ;OF THE PROGRAM
IDT 005A'ARG                  .
.ENDM                         .
    .                         .
    .                     ;WHERE 005A IS THE UPDATE
    .                     ;VERSION OF THE PROGRAM
OUT  \ID                  ;AND ARG INDICATES THE
                          ;CONDITIONAL ASSEMBLY VERSION.
```

The above macro call expands to:

```
        .ASCII  /005AXX/
```

where XX is the conditional value of ID.

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

6.3.4 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 .IFB and .IFNB can be used within the macro to detect unnecessary arguments.

A macro can be defined with no arguments.

6.3.5 Automatically Created Symbols

MACll can be made to 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 3.5.) Such local symbols are created by the Assembler in numerical order; i.e.:

```
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 (resulting in a multiply-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 causes a new local symbol block to be initialized.

The macro processor creates a local symbol on each call of a macro whose definition contains a dummy argument preceded by the ? character. For example:

```
        .MACRO  ALPHA  A,?B
        TST     A
        BEQ     B
        ADD     #5,A
B:
        .ENDM
```

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 expansions of the macro ALPHA above.

GENERATE A LOCAL SYMBOL FOR MISSING ARGUMENT:

```
        ALPHA %1
        TST   %1
        BEQ   64$
        ADD    #5,%1
64$:
```

DO NOT CREATE A LOCAL SYMBOL:

```
        ALPHA %2,XYZ
        TST   %2
        BEQ   XYZ
        ADD   #5,%2
XYZ:
```

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

6.3.6 Concatenation

The apostrophe or single quote character (') operates as a legal separating character in macro definitions. An ' character which 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:

```
        .MACRO DEF A,B,C
A'B:    .ASCIZ  /C/
        .WORD  ''A'''B
        .ENDM
```

When this macro is called:

```
DEF X,Y,<MAC11>
```

it expands as follows:

```
XY:     .ASCIZ  /MAC11/
        .WORD  'X'Y
```

In the macro definition, the scan terminates upon finding the first ' character. Since A is a dummy argument, the ' is removed. The scan resumes with B, notes B as another dummy argument and concatenates the two dummy arguments. The third dummy argument is noted as going into the operand of the .ASCIZ directive. On the next line (this example is purely for illustrative purposes) the argument to .WORD is seen as follows: The scan begins with a ' character. Since it is neither preceded nor followed by a dummy argument, the ' character remains in the macro definition. The scan then encounters the second ' character which is followed by a dummy argument and is discarded. The scan of the argument A terminated upon encountering the second ' which is also discarded since it follows a dummy argument. The next ' character is neither preceded nor followed by a dummy argument and remains in the macro expansion. The

last ' character is followed by another dummy argument and is discarded. (Note that the five ' characters were necessary to generate two ' characters in the macro expansion.)

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

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, and is of the form:

```
label:  .NARG    symbol
```

where:

label — is an optional statement label

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

This directive can occur only within a macro definition.

The .NCHR directive enables a program to determine the number of characters in a character string, and is of the form:

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

where:

labed — is an optional statement label.

symbol — is any legal symbol which is 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 which should only be enclosed in angle brackets if it contains a legal separator. A semi-colon also terminates the character string.

This directive can occur anywhere in a MAC11 program.

The .NTYPE directive enables the macro being expanded to determine the addressing mode of any argument, and is of the form:

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

where:

label is an optional statement label.

symbol is any legal symbol, the low-order 6-bits of which are 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 6.3.

This directive can occur only within a macro definition. An example of .NTYPE usage in a macro definition is shown below:

```
.MACRO SAVE                         ARG
.NTYPE SYM,ARG
.IF        EQ,SYM&70
MOV        ARG,TEMP                 ;REGISTER MODE
.IFF
MOV        #ARG,TEMP                ;NON-REGISTER MODE
.ENDC
.ENDM
```

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 string to be output to the command device. The text string is terminated by a line terminator.

Upon encountering a .ERROR directive anywhere in a MAC11 program, the Assembler outputs a single line containing:

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, and
4. the text string specified.

For example:

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

causes a line similar to the following to be output:

```
512  5642  000076        ;UNACCEPTABLE MACRO ARGUMENT
```

This message is being used to indicate an inability of the subject macro to cope with the argument A which is detected as being indexed deferred addressing mode (mode 70) with the stack pointer (%6) used as the index register.

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 which 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 is of the form:

```
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 which 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.

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 (for example, the .MEXIT statement is allowed in an indefinite repeat block). Indefinite repeat arguments follow the same rules as macro arguments.

```
1                                       .TITLE  IRPTST
2                                       .LIST   MD,MC,ME
                 000000 R0=% 00
                 000001 R1=% 01
                 000002 R2=% 02
                 000003 R3=% 03
                 000004 R4=% 04
                 000005 R5=% 05
                 000006 R6=% 06
                 000007 R7=% 07
                 000006 SP=% 06
                 000007 PC=% 07
                 177776 PSW= 0177776
                 177570 SWR= 0177570
3       000000   012/00          MOV     #TABLE,R0
                 000050
4
5                               .IRP    X,<A,B,C,D,E,F>
6
7                               MOV     X,(R0)+
8
9                               .ENDM

        00004    016720          MOV     A,(R0)+
                 000032


        00010    016720          MOV     B,(R0)+
                 000030


        00014    016720          MOV     C,(R0)+
                 000026


        00020    016720          MOV     D,(R0)+
                 000024


        00024    016720          MOV     E,(R0)+
                 000022


        00030    016720          MOV     F,(R0)+
                 000020
12
13                              .IRPC   X,ABCDEF
14
15                              .ASCII  /X/
16
17                              .ENDM

        00034        101         .ASCII  /A/

        00035        102         .ASCII  /B/

        00036        103         .ASCII  /C/

        00037        104         .ASCII  /D/
```

```
     00040    105          .ASCII  /E/

     00041    106          .ASCII  /F/

18
19
20   00042 041101 A:       .WORD   "AB
21   00044 041502 B:       .WORD   "BC
22   00046 042103 C:       .WORD   "CD
23   00050 042504 D:       .WORD   "DE
24   00052 043105 E:       .WORD   "EF
25   00054 043506 F:       .WORD   "FG
26   00056        TABLE:   .BLKW   6
27
28        000001           .END
```

Figure 6-1
.IRP and .IRPC 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. Terminators for the string are: space, comma, tab, carriage return, line feed, and semi-colon.

6.7 REPEAT BLOCK: .REPT

Occasionally it is useful to duplicate a block of code a number of times in-line with other source code. This is performed by creating a repeat block of the 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. Where expr<0, 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, 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.

P A R T I V

OPERATING PROCEDURES

This part of the manual describes the operation of the MACll Assembler, its input files and their formats, and the variations of the command string to the Assembler.

CHAPTER 7

OPERATING PROCEDURES

This MAC11 Assembler assembles one ASCII source file containing MAC11 statements at a time into a single absolute binary output file. The output of the Assembler consists of an absolute binary file on a paper tape, and an assembly listing followed by the symbol table listing on the device assigned to .DAT-12.

7.1 LOADING MAC11

MAC11 is loaded under DOS-15 by typing:

```
$MAC11      (followed by a carriage return or altmode)
```

(Characters printed by the system are underlined to differentiate them from characters printed 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  V1A
>
```

7.2 COMMAND INPUT STRING

In response to the > printed by the Assembler, the user types the switch options followed by the input filename; the switch options and the filename are separated by a '←'. Command input can be terminated by a carriage return to restart MAC11, or by an altmode to return to DOS-15 at the end of assembly:

```
>SW←FILNAM
```

where:

SW — is the switch option(s); can be null (for plain assembly,) or 'B' (for binary output) or 'L' (for listing) or both.

FILNAM — is the input filename extension or filename from .DAT-11. Default extension is 'SRC'. The filename can consist of up to six characters followed by a

space(s) and not more than a 3-character extension (additional characters cause the message 'NAME ERROR/TOO LONG' to be printed on the command inp3t device). All of the legal printing characters can be used in any order. The first non-space character to be typed after the first left-arrow (←) is recognized as the first character of the filename. Similarly, the first non-space character after the filename (other than carriage return or altmode) is recognized as the first character of the extension.

Examples:

>←FILNAM	plain assembly of a file called 'filnam SRC', and restart MAC11
>←FILNAM EXT	plain assembly of a file called 'FILNAM EXT', and restart MAC11.
>B←FILNAM EXT (ALT)	assemble 'FILNAM EXT' to obtain an absolute binary output on a paper tape and return to DOS-15 monitor.
>L←FILNAM EXT (ALT)	assemble 'FILNAM EXT' to obtain a listing output on .DAT-12 and return to DOS-15 monitor.
>LB←X1Y2 E0	assemble 'X1Y2 E0 to obtain an absolute binary output on a paper tape and a listing output on .DAT-12 and restart MAC11.

If an error is made in typing the command string, typing the RUBOUT key erases the immediately-preceding character. Repeated typing of the RUBOUT key erases one character for each RUBOUT up to the beginning of the line. Typing CTRL/U erases the entire line.

A syntactical error detected in the command string causes the Assembler to print a ? character. The Assembler then reprints the > character and waits for a new command string to be entered. If the input file is not found or name and/or extension is illegal, the message:

NAME ERROR/TOO LONG

is printed.

```
MACRO   V001                                    MACRO V000A1
OBJECT CODE HANDLERS

1                                       .SBTTL  OBJECT CODE HANDLERS
2
3 012026                ENDP1:                                  ;END OF PASS HANDLER
4 012026                        CALL    SETMAX
  012026 004767                 JSR     PC,SETMAC
         174240
5 012062 005767                 TST     PASS                    ;PASS ONE?
         000000'
6 012036 001142                 BNE     ENDP2                   ;BRANCH IF PASS 2
7 012040                                ENTOVR                  4
8 012040 005767                 TST     OBJLNK                  ;PASS ONE, ANY OBJECT?
         001416'
9 012044 001517                 BEQ     30$                     ;  NO
10 12046 012767                 MOV     #BLK[01,BLKTYP          ;SET BLOCK TYPE1 1
         000001
         000542'
11 12054                        CALL    OBJINI                  ;INIT THE POINTERS
   12054 004767                 JSR     PC,OBJINI
         001542
12 12060 012701                 MOV     #PRGTTL,R1              ;SET "FROM" INDEX
         000050'
13 12064 016702                 MOV     RLDPNT,R2               ;  AND "TO" INDEX
         000549'
14 12070                        CALL    GSDDMP                  ;OUTPUT GSD BLOCK
   12070 004767                 JSR     PC,GSDDMP
         000660
15 12074 005046                 CLR     -(SP)                   ;INIT FOR SECTOR SCAN
16 12076 012667         10$:    MOV     (SP)+,ROLUPD            ;SET SCAN MARKER
         000006'

17 12102                        NEXT    SECROL                  ;GET THE NEXT SECTOR
   12102 012700                 MOV     #SECROL,R0
       000010
   12106 004767                 JSR     PC,NEXT
         005400
18 12112 001450                 BEQ     20$                     ;BRANCH IF THROUGH
19 12114 016746                 MOV     ROLUPD,-(SP)            ;SAVE MARKER
         000006'
21 12124 011105                 MOV     (R1),R5                 ;SAVE SECTOR
22 12126 042705                 BIC     #377,R5                 ;ISOLATE IT
         000377
23 12132 000305                 SWAB    R5                      ;  AND PLACE IN RIGHT
24 12134 042711                 BIC     #-1-<RELFLG>,(R1)       ;CLEAR ALL BUT REL BIT
         177737
25 12140 052721                 BIS     #<GSDT01>+DEFFLG,(R1)+  ;SET TO TYPE 1, DEFINED
         000410
26 12144 010521                 MOV     R5,(R1)+                ;ASSUME ABS
27 12146 001401                 BEQ     11$                     ;  OOPS!
28 12150 011141                 MOV     (R1),-(R1)              ;  REL, SET MAX
```

Figure 7-1
Assembly Listing

```
29 12152 005067  11$:    CLR     ROLUPD          ;SET FOR INNER SCAN
         000006'
30 12156 012701  12$      MOV     #SYMBOL,R1
         000002'
31 12162                  CALL    GSDDMP          ;OUTPUT THIS BLOCK
   12162 004767           JSR     PC,GSDDMP
         000566
32 12166         13$:     NEXT    SYMBOL          ;FETCH THE NEXT SYMBOL
   12166 012700           MOV     #SYMBOL,R0
         000000
   12172 004767           JSR     PC,NEXT
         005314
33 12    001737           BEQ     10$             ;  FINISHED WITH THIS GUY
34 12200 032767           BIT     #GLBFLG,MODE    ;GLOBAL?
         000100
         000036'
35 12206 001767           BEQ     13$             ;  NO
36 12210 126705           CMPB    SECTOR,R5       ;YES, PROPER SECTOR?
         000007'
37 12214 001364           BNE     13$             ;  NO
38 12216 042767           BIC     #-1-<DEFFLG!RELFLG!GLBFLG>,MODE ;CLEAR MOST
         177627
         000006'
39 12224 052767           BIS     #GSDT04,MODE    ;SET TYPE 4
         002000
         000005'
40 12232 000751           BR      12$             ;OUTPUT IT
```

Figure 7-1 (Cont.)
Assembly Listing

APPENDIX A

MAC11 CHARACTER SETS

A.1 ASCII CHARACTER SET

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
0	000	NUL	NULL, TAPE FEED, CONTROL/SHIFT/P.
1	001	SOH	START OF HEADING: ALSO SOM, START OF MESSAGE, CONTROL/A.
1	002	STX	START OF TEXT; ALSO EOA, END OF ADDRESS, CONTROL/B.
0	003	ETX	END OF TEXT; ALSO EOM, END OF MESSAGE, CONTROL/C.
1	004	EOT	END OF TRANSMISSION (END); SHUTS OFF TWX MACHINES, CONTROL/D.
0	005	ENQ	ENQUIRY (ENQRY); ALSO WRU, CONTROL/E.
0	006	ACK	ACKNOWLEDGE; ALSO RU, CONTROL/F
1	007	BEL	RINGS THE BELL. CONTROL/G.
1	010	BS	BACKSPACE; ALSO FEO, FORMAT EFFECTOR. BACKSPACES SOME MACHINES. CONTROL/H.
0	011	HT	HORIZONTAL TAB. CONTROL/I.
0	012	LF	LINE FEED OR LINE SPACE (NEW LINE); ADVANCES PAPER TO NEXT LINE, DUPLICATED BY CONTROL/J.
1	013	VT	VERTICAL TAB (VTAB). CONTROL/K.
0	014	FF	FORM FEED TO TOP OF NEXT PAGE (PAGE). CONTROL/L.
1	015	CR	CARRIAGE RETURN TO BEGINNING OF LINE. DUPLICATED BY CONTROL/M.
1	016	SO	SHIFT OUT; CHANGES RIBBON COLOR TO RED. CONTROL/N.
0	017	SI	SHIFT IN; CHANGES RIBBON COLOR TO BLACK. CONTROL/O.
1	020	DLE	DATA LINK ESCAPE. CONTROL/B (DC0).
0	021	DC1	DEVICE CONTROL 1, TURNS TRANSMITTER (READER) ON, CONTROL/Q (X ON).
0	022	DC2	DEVICE CONTROL 2, TURNS PUNCH OR AUXILIARY ON. CONTROL/R (TAPE, AUX ON).
1	023	DC3	DEVICE CONTROL 3, TURNS TRANSMITTER (READER) OFF, CONTROL/S (X OFF).
0	024	DC4	DEVICE CONTROL 4, TURNS PUNCH OR AUXILIARY OFF. CONTROL/T (AUX OFF).
1	025	NAK	NEGATIVE ACKNOWLEDGE; ALSO ERR, ERROR. CONTROL/U.
1	026	SYN	SYNCHRONOUS FILE (SYNC). CONTROL/V.
0	027	ETB	END OF TRANSMISSION BLOCK; ALSO LEM, LOGICAL END OF MEDIUM. CONTROL/W.

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
0	030	CAN	CANCEL (CANCL). CONTROL/X.
1	031	EM	END OF MEDIUM. CONTROL/Y.
1	032	SUB	SUBSTITUTE. CONTROL/Z.
1	033	ESC	ESCAPE. CONTROL/SHIFT/K.
1	034	FS	FILE SEPARATOR. CONTROL/SHIFT/L.
0	035	GS	GROUP SEPARATOR. CONTROL/SHIFT/M.
0	036	RS	RECORD SEPARATOR. CONTROL/SHIFT/N.
1	037	US	UNIT SEPARATOR. CONTROL/SHIFT/O.
1	040	SP	SPACE.
0	041	!	
0	042	"	
1	043	#	
0	044	$	
1	045	%	
1	046	&	
0	047	'	ACCENT ACUTE OR APOSTROPHE.
0	050	(	
1	051	)	
1	052	*	
0	053	+	
1	054	,	
0	055	-	
0	056	.	
1	057	/	
0	060	0	
1	061	1	
1	062	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	067	7	
1	070	8	
0	071	9	
0	072	:	
1	073	;	
0	074	<	
1	075	=	
1	076	>	
0	077	?	
1	100	@	
0	101	A	
0	102	B	
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	
0	110	H	
1	111	I	
1	112	J	

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
0	113	K	
1	114	L	
0	115	M	
0	116	N	
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	
0	131	Y	
0	132	Z	
1	133	[	SHIFT/K.
0	134	\	SHIFT/L.
1	135	]	SHIFT/M.
1	136	↑	
0	137	←	
0	140	'	ACCENT GRAVE.
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	
1	171	y	
1	172	z	
0	173		
1	174		
0	175		THIS CODE GENERATED BY ALTMODE.
0	176		THIS CODE GENERATED BY PREFIX KEY (IF PRESENT).
1	177	DEL	DELETE, RUBOUT.

A.2 RADIX-50 CHARACTER SET

Character	ASCII Octal Equivalent	Radix-50 Equivalent
space	40	0
A-Z	101 - 132	1 - 32
$	44	33
.	56	34
unused		35
0 = 9	60 = 71	36 = 47

The maximum Radix-50 value is, thus,

47*50(2) + 47*50 + 47 = 174777

The following table provides a convenient means of translating between the ASCII character set and its Radix-50 equivalents. For example, given the ASCII string X2B, the Radix-50 equivalent is (arithmetic performed in octal):

```
  X = 113000
  2 = 002400
  B = 000002
X2B = 115402
```

Single Char. or First Char.		Second Character		Third Character	
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000015
N	053600	N	001060	N	000016
O	056700	O	001130	O	000017
P	062000	P	001200	P	000020
Q	065100	Q	001250	Q	000021
R	070200	R	001320	R	000022
S	073300	S	001370	S	000023
T	076400	T	001440	T	000024
U	101500	U	001510	U	000025
V	104600	V	001560	V	000026
W	107700	W	001630	W	000027
X	113000	X	001700	X	000030
Y	116100	Y	001750	Y	000031
Z	121200	Z	002020	z	000032
$	124300	$	002070	$	000033
.	127400	.	002140	.	000034
unused	132500	unused	002210	unused	000035
0	135600	0	002260	0	000036
1	140700	1	002330	1	000037
2	144000	2	002400	2	000040
3	147100	3	002450	3	000041
4	152200	4	002520	4	000042
5	155300	5	002570	5	000043
6	160400	6	002640	6	000044
7	163500	7	002710	7	000045
8	166600	8	002760	8	000046
9	171700	9	003030	9	000047

APPENDIX B

MAC11 ASSEMBLY LANGUAGE AND ASSEMBLER

B.1 SPECIAL CHARACTERS

Character	Function
form feed	Source line terminator
line feed	Source line terminator
carriage return	Formatting character
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 autoincrement indicator
-	Arithmetic subtraction operator or autodecrement indicator
*	Arithmetic multiplication operator
/	Arithmetic division operator
&	Logical AND operator
!	Logical OR operator
"	Double ASCII character indicator

' (apostrophe)	Single ASCII line indicator
.	Assembly location counter
<	Initial argument indicator
>	Terminal argument indicator
↑	Universal unary operator Argument indicator
\	MACRO numeric argument indicator

B.2 ADDRESS MODE SYNTAX

n is an integer between 0 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 0 to 7.

Format	Address Mode Name	Address Mode Number	Meaning
R	Register	0n	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 Autoincrement	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 Autodecrement	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 added to ER gives the pointer to 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.

B.3 INSTRUCTIONS

The instructions which follow are grouped according to the operands they take and the bit patterns of their op-codes.

In the instruction-type format specification, the following symbols are used:

OP	Instruction mnemonic
R	Register expression
E	Expression
ER	Register expression or expression 0<ER<7
A	General address specification

In the representation of op-codes, the following symbols are used:

SS	Source operand specified by a 6-bit address mode.
DD	Destination operand specified by a 6-bit address mode.
XX	8-bit offset to a location (branch instructions).
R	Integer between 0 and 7 representing a general register.

Symbols used in the description of instruction operands are:

SE	Source Effective Address
DE	Destination Effective address
| |	Absolute value of
( )	Contents of
	Becomes

The condition codes in the processor status word (PS) are affected by the instructions. These condition codes are represented as follows:

N	Negative bit:	set if the result is negative
Z	Zero bit:	set if the result is zero
V	oVerflow bit:	set if the operation caused an overflow
C	Carry bit:	set if the operation caused a carry.

In the representation of the instruction's effect on the condition codes, the following symbols are used:

*	Conditionally set
-	Not affected
0	Cleared
1	Set

To set conditionally means to use the instruction's result to determine the state of the code (see the PDP-11 Processor Handbook).

Logical operations are represented by the following symbols:

!	Inclusive OR
⊕	Exclusive OR
&	AND
-	(used over a symbol) NOT (i.e., 1's complement)

B.3.1 Double-Operand Instructions

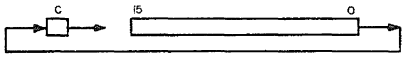
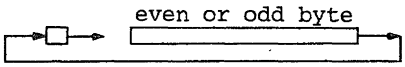
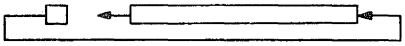
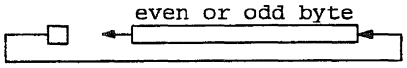
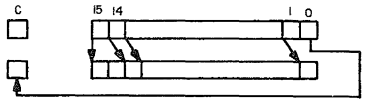
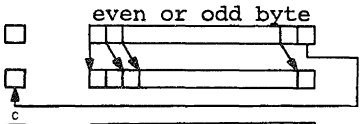
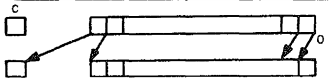
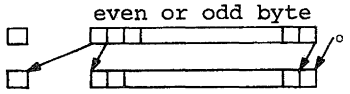
Instruction type format: Op A,A

				Status Word Condition Codes			
Op-Code	Mnemonic	Stands for	Operation	N	Z	V	C
01SSDD	MOV	MOVe	(SE) → (DE)	*	*	0	-
11SSDD	MOVB	MOVe Byte					
02SSDD	CMP	CoMPare	(SE) - (DE)	*	*	*	*
12SSDD	CMPB	CoMPare Byte					
03SSDD	BIT	BIt Test	(SE) & (DE)	*	*	0	-
13SSDD	BITB	BIt Test Byte					
04SSDD	BIC	BIt Clear	(SE) & (DE) → DE	*	*	0	-
14SSDD	BICB	BIt Clear Byte					
05SSDD	BI"	BIt Set	(SE) ! (DE) → DE	*	*	0	-
15SSDD	BISB	BIt Set Byte					
06SSDD	ADD	ADD	(SE) + (DE) → DE	*	*	*	*
16SSDD	SUB	SUBtract	(DE) - (SE) → E	*	*	*	*

B.3.2 Single-Operand Instructions

Instruction-type format: Op A

				Status Word Condition Codes			
Op-Code	Mnemonic	Stands for	Operation	N	Z	V	C
0050DD	CLR	CLear	0 → DE	0	1	0	0
1050DD	CLRB	CLear Byte					
0051DD	COM	COMplement	(DE) → DE	*	*	0	1
1051DD	COMB	COMplement Byte					

0052DD 1052DD	INC INCB	INCrement INCrement Byte	(DE)+1 DE	*	*	*	-
0053DD 1053DD	DEC DECB	DECrement DECrement Byte	(DE)-1 DE	*	*	*	-
0054DD 1054DD	NEG NEGB	NEGate NEGate Byte	(DE)+1 DE	*	*	*	*
0055DD 1055DD	ADC ADCB	ADd Carry ADd Carry Byte	(DE)+(C) DE	*	*	*	*
0056DD 1056DD	SBC SBCB	SuBtract Carry SuBtract Carry Byte	(DE)-(C) DE	*	*	*	*
0057DD 1057DD	TST TSTB	TeST TeST Byte	(DE)-0 DE	*	*	0	0
0060DD	ROR	ROtate Right		*	*	*	*
1060DD	RORB	ROtate Right Byte		*	*	*	*
0061DD	ROL	ROtate Left		*	*	*	*
1061DD	ROLB	ROtate Left Byte		*	*	*	*
0062DD	ASR	Arithmetic Shift Right		*	*	*	*
1062DD	ASRB	Arithmetic Shift Right Byte		*	*	*	*
0063DD	ASL	Arithmetic Shift Left		*	*	*	*
1063DD	ASLB	Arithmetic Shift Left Byte		*	*	*	*
0001DD	JMP	JuMP	DE PC	-	-	-	-
0003DD	SWAB	SWAp Bytes		*	*	0	0
0067DD	SXT	Sign eXTend	0 DE if N bit clear -1 DE if N bit is set	-	*	-	-
				FN	FZ	FV	FC
0707DD	NEGD	NEGate Double	-(FDE) FDE	*	*	0	0

1704DD	CLRD	CLeaR Double	0 FDE	0	1	0	0
1705DD	TSTD	TeST Doubel	(FDE)-0 FDE	*	*	0	0
1706DD	ABSD	make ABSolute	FDE FDE	0	*	0	0

B.3.3 Operate Instructions

Instruction-Type format: Op

Op-Code	Mnemonic	Stands for	Operation	N	Z	V	C
000000	HALT	HALT	The computer stops all functions.	-	-	-	-
000001	WAIT	WAIT	The computer stops and waits for an interrupt.	-	-	-	-
000002	RTI	ReTurn from Interrupt	The PC and PS are popped off the SP stack: ((SP))→PC (SP)+2→SP ((SP))→PS (SP)+2→SP RTI is also used to return from a trap.	*	*	*	*
000005	RESET	RESET	Returns all I/O devices to power-on status.	-	-	-	-
000241	CLC	CLear Carry bit	0→C	-	-	-	0
000261	SEC	SEt Carry bit	1→C	-	-	-	1
000242	CLV	CLear oVerflow bit	0→V	-	-	0	-
000262	SEV	SEt oVerflow bit	1→V	-	-	1	-
000244	CLZ	CLear Zero bit	0→Z	-	0	-	-
000264	SEZ	SEt Zero bit	1→Z	-	1	-	-
000250	CLN	CLear Negative bit	0→N	0	-	-	-
000270	SEN	SEt Negative bit	1→N	1	-	-	-
000243	CVC	Clear oVerflow and Carry bits	0→V	-	-	0	0
000254	CNZ	Clear Negative and Zero bits	0→N 0→Z	0	0	-	-

000257	CCC	Clear all Condition Codes	0→N 0→Z 0→V 0→C	0	0	0	0
000277	SCC	Set all Condition Codes	1→N 1→Z 1→V 1→C	1	1	1	1
000240	NOP	No OPeration		-	-	-	-

B.3.4 Trap Instructions

Instruction-type format: Op or Op E where 0 < E < 377(8)
*OP (only)

				Status Word Condition Codes			
Op-Code	Mnemonic	Stands for	Operation	N	Z	V	C
000003	BPT	BreakPoint Trap	Trap to location 14. This is used to call ODT.	*	*	*	*
*000004	IOT	Input/Output Trap	Trap to location 20. This is used to call IOX.	*	*	*	*
104000-104377	EMT	EMulator Trap	Trap to location 30. This is used to call system programs.	*	*	*	*
104400-104777	TRAP	TRAP	Trap to location 34. This is used to call any routine desired by the programmer.	*	*	*	*

B.3.5 Branch Instructions

Instruction-type format: Op E where -128(10) < (E-.-2)/2 < 127(10)

Op-Code	Mnemonic	Stands for	Condition to be met if branch is to occur
0004XX	BR	BRanch always	
0010XX	BNE	Branch if Not Equal (to zero)	Z=0
0014XX	BEQ	Branch if EQual (to zero)	Z=1
0020XX	BGE	Branch if Greater than or Equal (to zero)	N ⊕ V=0
0024XX	BLT	Branch if Less than (zero)	N ⊕ V=1
0030XX	BGT	Branch if Greater than (zero)	Z ∨ (N ⊕ V)=0
0034XX	BLE	Branch if Less than or equal (to zero)	Z ∨ (N ⊕ V)=1
1000XX	BPL	Branch if PLus	N=0
1004XX	BMI	Branch if MInus	N=1
1010XX	BHI	Branch if HIgher	C ∨ Z=0
1014XX	BLOS	Branch if LOwer or Same	C ∨ Z=1
1020XX	BVC	Branch if oVerflow Clear	V=0
1024XX	BVS	Branch if oVerflow Set	V=1
1030XX	BCC (or BHIS)	Branch if Carry Clear (or Branch if Higher or Same)	C=0
1034XX	BCS (or BLOS)	Branch if Carry Set (or Branch if Lower	C=1

B.3.6 Register Destination

Instruction type format: Op ER,A

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
004RDD	JSR	Jump to SubRoutine	Push register on the SP stack, put the PC in the register. DE TEMP (TEMP= temporary storage register internal to processor.) (SP)-2 SP (REG) (SP) (PC) REG (TEMP) PC	-	-	-	-

The following instruction is available only on the PDP-11/45:

074RDD	XOR	eXclusive OR	(R) ! DE DE	*	*	0	-

B.3.7 Subroutine Return

Instruction type format: Op ER

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
00020R	RTS	ReTurn from Subroutine	Put register in PC and pop old contents from SP stack into register	-	-	-	-

B.4 ASSEMBLER DIRECTIVES

Form	Operation	Described in Manual Section
'	A single-quote character (apostrophe) followed by one ASCII character generates a word containing the 7-bit ASCII representation of the character in the low-order byte and zero in the high order byte.	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.	5.3.3
↑Bn	Temporary radix control; causes the number n to be treated as a binary number.	5.4.2
↑Cn	Creates a word containing the one's complement of n.	5.6.2
↑Dn	Temporary radix control; causes the number n to be treated as a decimal number.	5.4.2
↑On	Temporary radix control; causes the number n to be treated as an octal number.	5.4.2
.ASCII string	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters) one character per byte.	5.3.4
.ASCIZ string	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.	5.3.5
.BLKB exp	Reserves a block of storage space exp bytes long.	5.5.3

FORM	Operation	Described in Manual Section
.BLKW exp	Reserves a block of storage space exp words long.	5.5.3
.BYTE expl,exp2,...	Generates successive bytes of data containing the octal equivalent of the expression(s) specified.	5.3.1
.DSABL arg	Disables the assembler function specified by the argument.	5.2
.ENABL arg	Provides the assembler function specified by the argument.	5.2
.END .END exp	Indicates the physical end of source program. An optional argument specifies the transfer address.	5.7.1
.ENDC	Indicates the end of a conditional block.	5.11
.ENDM .ENDM symbol	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.	6.1.2
.ERROR exp,string	Causes a text string to be output to the command device containing the optional expression specified and the indicated text string.	6.5
.EVEN	Ensures that the assembly location counter contains an even address by adding 1 if it is odd.	5.5.1
.IF cond,arg1,arg2,...	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.	5.11
.IFF	Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested false.	5.11.1

Form	Operation	Described in Manual Section
.IFT	Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested true.	5.11.1
.IFTF	Appears only within a conditional block and indicates the beginning of a section of code to be unconditionally assembled.	5.11.1
.IIF cond,arg,statement	Acts as a 1-line conditional block where the condition is tested for the argument specified. The statement is assembled only if the condition tests true.	5.11.2
.IRP sym,<argl,arg2,...>	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).	6.6
.IRPC sym,string	Indicates the beginning of an indefinite repeat block in which the symbol specified takes on the value of successive characters in the character string.	6.6
.LIST .LIST arg	Without an argument, .LIST increments the listing level count by one. With an argument .LIST does not alter the listing level count but formats the assembly listing according to the argument specified.	5.1.1
.MACRO sym,argl,arg2,...	Indicates the start of a macro named sym containing the dummy arguments specified.	6.1.1
.MEXIT	Causes an exit from the current macro or indefinite repeat block.	6.1.3
.NARG symbol	Appears only within a macro definition and equates the specified symbol to the number of characters in the string (enclosed in delimiting characters).	6.4

Form	Operation	Described in Manual Section
.NCHR sym,string	Can appear anywhere in a source program; equates the symbol specified to the number of characters in the string (enclosed in delimiting characters).	6.4
.NLIST .NLIST arg	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.	5.1.1
.NTYPE sym,arg	Appears only in a macro definition and equates the low-order six bits of the symbol specified to the six-bit addressing mode of the argument.	6.4
.ODD	Ensures that the assembly location counter contains an odd address by adding 1 if it is even.	5.5.1
.PAGE	Causes the assembly listing to skip to the top of the next page.	5.1.6
.PRINT exp,string	Causes a text string to be output to the command device containing the optional expression specified and the indicated text string.	6.5
.RADIX n	Alters the current program radix to n, where n can be 2, 4, 8, or 10.	5.4.1
.RAD50 string	Generates a block of data containing the Radix-50 equivalent of the character string (enclosed in delimiting characters).	5.3.6
.REPT exp	Begins a repeat block. Causes the section of code up to the next .ENDM or or .ENDR to be repeated exp times.	6.7
.SBTTL string	Causes the string to be printed as part of the assembly listing page header. The string part of each .SBTTL directive is collected into	5.1.4

Form	Operation	Described in Manual Section
	a table of contents at the beginning of the assembly listing.	
.TITLE string	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.	5.1.3
.WORD expl,exp2,...	Generates successive words of data containing the octal equivalent of the expression(s) specified.	5.3.2

APPENDIX C

PERMANENT SYMBOL TABLE

```
PST       PERMANENT SYMBOL TABLE  MACRO V004A  PAGE 1


1                          .TITLE  PST    PERMANENT SYMBOL TABLE
2
3                          COPYRIGHT 1972  DIGITAL EQUIPMENT CORPORATION
4
5
6
7
8
9
10          000020 DR1=    200        ;DESTRUCTIVE REFERENCE IN FIRST
11          000100 DR2=    100        ;DESTRUCTIVE REFERENCE IN SECOND
12
13
14
15          000020 DFLGEV= 020        ;DIRECTIVE REQUIRES EVEN LOCATION
16          000010 DFLGBM= 010        ;DIRECTIVE USES BYTE MODE
17          000004 DFLCND= 004        ;CONDITIONAL DIRECTIVE
18          000002 DFLMAC= 002        ;MACRO DIRECTIVE
19
20
21
22                         .IIF DF X45,    XFLTG= 0
23                         .IIF DF XMACRO, XSMCAL= 0
24
25                         .MACRO  OPCDEF NAME, CLASS, VALUE, FLAGS, COND
26                         .IF NB  <COND>
27                         .IF DF  COND
28                         .MEXIT
29                         .ENDC
30                         .ENDC
31                         .RAD50  /NAME/
32                         .BYTE   FLAGS+0
33                         .GLOBL  OPCL'CLASS
34                         .BYTE   200+OPCL'CLASS
35                         .WORD   VALUE
36                         .ENDM
37
38                         .MACRO  DIRDEF NAME, FLAGS, COND
39                         .IF NB  <COND>
40                         .IF DF  COND
41                         .MEXIT
42                         .ENDC
43                         .ENDC
44                         .GLOBL  NAME
45                         .RAD50  /.'NAME/
46                         .BYTE   FLAGS+0
47                         .BYTE   0
48                         .WORD   NAME
49                         .ENDM
50
51  00000         PSTBAS:           ;BASE
```

```
1   000020          OPCDEF      <ADC  >,    01,    005500, DR1
2   000030          OPCDEF      <ADCB >,    01,    105500, DR1
3   000040          OPCDEF      <ADD  >,    02,    060000, DR2
4   000110          OPCDEF      <ASL  >,    01,    006300, DR1
5   000120          OPCDEF      <ASLB >,    01,    106300, DR1
6   000130          OPCDEF      <ASR  >,    01,    006200, DR1
7   000140          OPCDEF      <ASRB >,    01,    106200, DR1
8   000150          OPCDEF      <BCC  >,    04,    103000,
9   000160          OPCDEF      <BCS  >,    04,    103400,
10  000170          OPCDEF      <BEQ  >,    04,    001400,
11  000200          OPCDEF      <BGE  >,    04,    002000,
12  000210          OPCDEF      <BGT  >,    04,    003000,
13  000220          OPCDEF      <BHI  >,    04,    101000,
14  000230          OPCDEF      <BHIS >,    04,    103000,
15  000240          OPCDEF      <BIC  >,    02,    040000, DR2
16  000250          OPCDEF      <BICB >,    02,    140000, DR2
17  000260          OPCDEF      <BIS  >,    02,    050000, DR2
18  000270          OPCDEF      <BISB >,    02,    150000, DR2
19  000300          OPCDEF      <BIT  >,    02,    030000,
20  000310          OPCDEF      <BITB >,    02,    130000,
21  000320          OPCDEF      <BLE  >,    04,    003400,
22  000330          OPCDEF      <BLO  >,    04,    103400,
23  000340          OPCDEF      <BLOS >,    04,    101400,
24  000350          OPCDEF      <BLT  >,    04,    002400,
25  000360          OPCDEF      <BMI  >,    04,    100400,
26  000370          OPCDEF      <BNE  >,    04,    001000,
27  000400          OPCDEF      <BPL  >,    04,    100000,
28  000420          OPCDEF      <BR   >,    04,    000400,
29  000430          OPCDEF      <BVC  >,    04,    102000,
30  000440          OPCDEF      <BVS  >,    04,    102400,
31  000450          OPCDEF      <CCC  >,    00,    000257,
33  000470          OPCDEF      <CLC  >,    00,    000241,
34  000500          OPCDEF      <CLN  >,    00,    000250,
35  000510          OPCDEF      <CLR  >,    01,    005000, DR1
36  000520          OPCDEF      <CLRB >,    01,    105000, DR1
39  000550          OPCDEF      <CLV  >,    00,    000242,
40  000560          OPCDEF      <CLZ  >,    00,    000244,
```

```
1   000570                OPCDEF      <CMP  >,    02,    020000,
2   000600                OPCDEF      <CMPB >,    02,    120000,
                    CMZ   00          000254,
3   000630                OPCDEF      <COM  >,    01,    005100, DR1
4   000640                OPCDEF      <COMB >,    01,    105100, DR1
5   000650                OPCDEF      <DEC  >,    01,    005300, DR1
6   000660                OPCDEF      <DECB >,    01,    105300, DR1
7   000670                OPCDEF      <EMT  >,    06,    104000,
8   000730                OPCDEF      <HALT >,    00,    000000,
9   000740                OPCDEF      <INC  >,    01,    005200, DR1
10  000750                OPCDEF      <INCB >,    01,    105200, DR1
11  000760                OPCDEF      <IOT  >,    00,    000004,
12  000770                OPCDEF      <JMP  >,    01,    000100,
13  001000                OPCDEF      <JSR  >,    05,    004000, DR1
14  001010                OPCDEF      <MOV  >,    02,    010000, DR2
15  001230                OPCDEF      <MOVB >,    02,    110000, DR2
16  001240                OPCDEF      <NEG  >,    01,    005400, DR1
17  001320                OPCDEF      <NEGB >,    01,    105400, DR1
18  001330                OPCDEF      <NOP  >,    00,    000240,
19  001360                OPCDEF      <RESET>,    00,    000005,
20  001370                OPCDEF
```

```
1   001400          OPCDEF      <ROL  >,    01,    006100, DR1
2   001410          OPCDEF      <ROLB >,    01,    106100, DR1
3   001420          OPCDEF      <ROR  >,    01,    006000, DR1
4   001430          OPCDEF      <RORB >,    01,    106000, DR1
5   001440          OPCDEF      <RTI  >,    00,    000002,
6   001450          OPCDEF      <RTS  >,    03,    000200, DR1
7   001470          OPCDEF      <SBC  >,    01,    005600, DR1
8   001500          OPCDEF      <SBCB >,    01,    105600, DR1
9   001510          OPCDEF      <SCC  >,    00,    000277,
10  001520          OPCDEF      <SEC  >     00,    000261,
11  001530          OPCDEF      <SEN  >,    00,    000270,
12  001600          OPCDEF      <SEV  >,    00,    000262,
13  001610          OPCDEF      <SEZ  >,    00,    000264,
14  002020          OPCDEF      <SUB  >,    02,    160000,
15  002050          OPCDEF      <SWAB >,    01,    000300, DR1
16  002070          OPCDEF      <TRAP >,    06,    104400,
17  002100          OPCDEF      <TST  >,    01,    005700,
18  002110          OPCDEF      <TSTB >,    01,    105700,
19  002140          OPCDEF      <WAIT >,    00,    000001,
```

```
1   002160              DIRDEF          <ASCII>,        DFLGBM
2   002170              DIRDEF          <ASCIZ>,        DFLGBM
3   002210              DIRDEF          <BLKB >,
4   002220              DIRDEF          <BLKW >,        DFLGEV
5   002230              DIRDEF          <BYTE >,        DFLGBM
6   002250              DIRDEF          <DSABL>,
7   002260              DIRDEF          <ENABL>,
8   002270              DIRDEF          <END  >,
9   002300              DIRDEF          <ENDC >,        DFLCND
10  002310              DIRDEF          <ENDM >,        DFLMAC, XMACRO
11  002320              DIRDEF          <ENDR >,        DFLMAC, XMACRO
12  002340              DIRDEF          <ERROR>,
13  002350              DIRDEF          <EVEN >,
14  002420              DIRDEF          <IF   >,        DFLCND
15  002430              DIRDEF          <IFDF >,        DFLCND
16  002440              DIRDEF          <IFEQ >,        DFLCND
17  002450              DIRDEF          <IFF  >,        DFLCND
18  002460              DIRDEF          <IFG  >,        DFLCND
19  002470              DIRDEF          <IFGE >,        DFLCND
20  002500              DIRDEF          <IFGT >,        DFLCND
21  002510              DIRDEF          <IFL  >,        DFLCND
22  002520              DIRDEF          <IFLE >,        DFLCND
23  002530              DIRDEF          <IFLT >,        DFLCND
24  002540              DIRDEF          <IFNDF>,        DFLCND
25  002550              DIRDEF          <IFNE >,        DFLCND
26  002560              DIRDEF          <IFNZ >,        DFLCND
27  002570              DIRDEF          <IFT  >,        DFLCND
28  002600              DIRDEF          <IFTF >,        DFLCND
29  002610              DIRDEF          <IFZ  >,        DFLCND
30  002620              DIRDEF          <IIF  >,
31  002630              DIRDEF          <IRP  >,        DFLMAC, XMACRO
32  002640              DIRDEF          <IRPC >,        DFLMAC, XMACRO
33  002660              DIRDEF          <LIST >,
```

```
1   002670                DIRDEF          <MACR >,        DFLMAC, XMACRO
2   002700                DIRDEF          <MACRO>,        DFLMAC, XMACRO
3   002720                DIRDEF          <MEXIT>,        ,       XMACRO
4   002730                DIRDEF          <NARG >,        ,       XMACRO
5   002740                DIRDEF          <NCHR >,        ,       XMACRO
6   002750                DIRDEF          <NLIST >
7   002760                DIRDEF          <NTYPE>,        ,       XMACRO
8   002770                DIRDEF          <ODD  >,
9   003000                DIRDEF          <PAGE >,
10  003010                DIRDEF          <PRINT>,
11  003020                DIRDEF          <RADIX>,
12  003030                DIRDEF          <RAD50>,        DFLGEV
13  003040                DIRDEF          <REM  >,
14  003050                DIRDEF          <REPT >,        DFLMAC, XMACRO
15  003060                DIRDEF          <SBTTL >,
16  003070                DIRDEF          <TITLE >,
17  003100          WRDSYM:
18  003100                DIRDEF          <WORD >,        DFLGEV
19
20
21  003110          PSTTOP:                                 ,TOP LIMIT
22
23          000001          .END
```

APPENDIX D

ERROR MESSAGE SUMMARY

D.1 MAC11 ERROR CODES

MAC11 error codes are printed following a field of six asterisk characters and on the line preceding the source line containing the error. For example:

```
******A
26 00236  000002'     .WORD REL1+REL2
```

The addition of two relocatable symbols is flagged as an A error.

Error Code	Meaning
A	Addressing error. An address within the instruction is incorrect. Also may indicate a relocation error. This message does not necessarily reflect a coding error.
B	Bounding error. Instructions or word data are being assembled at an odd address in memory. The location counter is updated by +1.
D	Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.
E	End directive not found. (A listing is generated.)
I	Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing. The character is then ignored.
L	Line buffer overflow; i.e., input line greater than 132 characters. Extra characters on a line (more than 72(10)) are ignored.
M	Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a previously encountered label.
N	Number containing 8 or 9 has decimal point missing.
O	Op-code error. Directive out of context.
P	Phase error. A label's definition of value varies from one pass to another.
Q	Questionable syntax. There are missing arguments or the instruction scan was not completed or a carriage return was not immediately followed by a line feed or form feed.

R	Register-type error. An invalid use of or reference to a register has been made.
T	Truncation error. A number generated more than 16 bits of significance or an expression generated more than 8 bits of significance during the use of the .BYTE directive.
U	Undefined symbol. An undefined symbol was encountered during the evaluation of an expression. Relative to the expression, the undefined symbol is assigned a value of zero.
Z	Instruction which is not compatible among all members of the PDP-11 family (11/15, 11/20, and 11/45).

INDEX

HOW TO OBTAIN SOFTWARE INFORMATION

SOFTWARE NEWSLETTERS, MAILING LIST

The Software Communications Group, located at corporate headquarters in Maynard, publishes newsletters and Software Performance Summaries (SPS) for the various Digital products. Newsletters are published monthly, and contain announcements of new and revised software, programming notes, software problems and solutions, and documentation corrections. Software Performance Summaries are a collection of existing problems and solutions for a given software system, and are published periodically. For information on the distribution of these documents and how to get on the software newsletter mailing list, write to:

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

SOFTWARE PROBLEMS

Questions or problems relating to Digital's software should be reported to a Software Support Specialist. A specialist is located in each Digital Sales Office in the United States. In Europe, software problem reporting centers are in the following cities.

Reading, England	Milan, Italy
Paris, France	Solna, Sweden
The Hague, Holland	Geneva, Switzerland
Tel Aviv, Israel	Munich, West Germany

Software Problem Report (SPR) forms are available from the specialists or from the Software Distribution Centers cited below.

PROGRAMS AND MANUALS

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center.

Digital Equipment Corporation
Software Distribution Center
146 Main Street
Maynard, Massachusetts 01754

Digital Equipment Corporation
Software Distribution Center
1400 Terra Bella
Mountain View, California 94043

Outside of the United States, orders should be directed to the nearest Digital Field Sales Office or representative.

USERS SOCIETY

DECUS, Digital Equipment Computer Users Society, maintains a user exchange center for user-written programs and technical application information. A catalog of existing programs is available. The society publishes a periodical, DECUSCOPE, and holds technical seminars in the United States, Canada, Europe, and Australia. For information on the society and membership application forms, write to:

DECUS
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754

DECUS EUROPE
Digital Equipment Corporation
International (Europe)
P.O. Box 340
1211 Geneva 26
Switzerland

MAC11 Programming Language
(MAC11)
DEC-15-LMCMA-A-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form (see the HOW TO OBTAIN SOFTWARE INFORMATION page).

Did you find errors in this manual? If so, specify by page.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

If you do not require a written reply, please check here. ☐

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

printed in U.S.A.