MACll Programming Language
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:

<table>
<thead>
<tr>
<th>CDP</th>
<th>DIGITAL</th>
<th>INDAC</th>
<th>PS/8</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPUTER LAB</td>
<td>DNC</td>
<td>KA10</td>
<td>QUICKPOINT</td>
</tr>
<tr>
<td>COMSYST</td>
<td>EDGRIN</td>
<td>LAB-8</td>
<td>RAD-8</td>
</tr>
<tr>
<td>COMTEX</td>
<td>EDUSYSTEM</td>
<td>LAB-8/e</td>
<td>RSTS</td>
</tr>
<tr>
<td>DDT</td>
<td>FLIP CHIP</td>
<td>LAB-K</td>
<td>RSX</td>
</tr>
<tr>
<td>DEC</td>
<td>FOCAL</td>
<td>OMNIBUS</td>
<td>RTM</td>
</tr>
<tr>
<td>DECCOMM</td>
<td>GLC-8</td>
<td>OS/8</td>
<td>RT-11</td>
</tr>
<tr>
<td>DETAPE</td>
<td>IDAC</td>
<td>PDQ</td>
<td>SABR</td>
</tr>
<tr>
<td>DIBOL</td>
<td>IDACS</td>
<td>PHA</td>
<td>TYPESET 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>UNIBUS</td>
</tr>
</tbody>
</table>
# CONTENTS

## PART I

**INTRODUCTION TO MAC11**

<table>
<thead>
<tr>
<th>CHAPTER 1</th>
<th>FUNDAMENTALS OF PROGRAMMING THE PDP-11</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>MODULAR PROGRAMMING</td>
<td>1-1</td>
</tr>
<tr>
<td>1.1.1</td>
<td>Commenting PDP-11 Assembly Language Programs</td>
<td>1-4</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Localized Register Usage</td>
<td>1-4</td>
</tr>
<tr>
<td>1.1.3</td>
<td>Conditional Assemblies</td>
<td>1-6</td>
</tr>
<tr>
<td>1.2</td>
<td>REENTRANT CODE</td>
<td>1-6</td>
</tr>
<tr>
<td>1.3</td>
<td>PREFERRED ADDRESSING MODES</td>
<td>1-6</td>
</tr>
<tr>
<td>1.4</td>
<td>PARAMETER ASSIGNMENTS</td>
<td>1-7</td>
</tr>
<tr>
<td>1.5</td>
<td>SPACE VS. TIMING TRADEOFFS</td>
<td>1-7</td>
</tr>
<tr>
<td>1.5.1</td>
<td>Trap Handler</td>
<td>1-7</td>
</tr>
<tr>
<td>1.5.2</td>
<td>Register Increment</td>
<td>1-8</td>
</tr>
<tr>
<td>1.6</td>
<td>CONDITIONAL BRANCH INSTRUCTIONS</td>
<td>1-8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 2</th>
<th>SOURCE PROGRAM FORMAT</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>STATEMENT FORMAT</td>
<td>2-1</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Label Field</td>
<td>2-2</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Operator Field</td>
<td>2-3</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Operand Field</td>
<td>2-3</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Comment Field</td>
<td>2-3</td>
</tr>
<tr>
<td>2.2</td>
<td>FORMAT CONTROL</td>
<td>2-4</td>
</tr>
</tbody>
</table>

## PART II

**DETAILS ON PROGRAMMING IN MAC11**

<table>
<thead>
<tr>
<th>CHAPTER 3</th>
<th>SYMBOLS AND EXPRESSIONS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>CHARACTER SET</td>
<td>3-1</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Separating and Delimiting Characters</td>
<td>3-2</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Illegal Characters</td>
<td>3-4</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Operator Characters</td>
<td>3-4</td>
</tr>
<tr>
<td>3.2</td>
<td>MAC11 SYMBOLS</td>
<td>3-5</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Permanent Symbols</td>
<td>3-5</td>
</tr>
<tr>
<td>3.2.2</td>
<td>User-Defined and MACRO Symbols</td>
<td>3-5</td>
</tr>
<tr>
<td>3.3</td>
<td>DIRECT ASSIGNMENTS</td>
<td>3-6</td>
</tr>
<tr>
<td>3.4</td>
<td>REGISTER SYMBOLS</td>
<td>3-7</td>
</tr>
<tr>
<td>3.5</td>
<td>LOCAL SYMBOLS</td>
<td>3-8</td>
</tr>
<tr>
<td>3.6</td>
<td>ASSEMBLY LOCATION COUNTER</td>
<td>3-10</td>
</tr>
<tr>
<td>3.7</td>
<td>NUMBERS</td>
<td>3-11</td>
</tr>
<tr>
<td>3.8</td>
<td>TERMS</td>
<td>3-11</td>
</tr>
<tr>
<td>3.9</td>
<td>EXPRESSIONS</td>
<td>3-12</td>
</tr>
</tbody>
</table>
CHAPTER 4  ADDRESSING MODES

4.1 REGISTER MODE  4-2
4.2 REGISTER DEFERRED MODE  4-2
4.3 AUTOINCREMENT MODE  4-2
4.4 AUTOINCREMENT DEFERRED MODE  4-3
4.5 AUTODECREMENT MODE  4-3
4.6 AUTODECREMENT DEFERRED MODE  4-3
4.7 INDEX MODE  4-4
4.8 INDEX DEFERRED MODE  4-4
4.9 IMMEDIATE MODE  4-4
4.10 ABSOLUTE MODE  4-5
4.11 RELATIVE MODE  4-5
4.12 RELATIVE DEFERRED MODE  4-6
4.13 TABLE OF MODE FORMS AND CODES  4-6
4.14 BRANCH INSTRUCTION ADDRESSING  4-7

PART III
MACII ASSEMBLER DIRECTIVES

CHAPTER 5  GENERAL ASSEMBLER DIRECTIVES

5.1 LISTING CONTROL DIRECTIVES  5-1
5.1.1 .LIST and .NLIST  5-1
5.1.2 Page Headings  5-5
5.1.3 .TITLE  5-5
5.1.4 .SBTTL  5-5
5.1.5 Page Ejection  5-7
5.2 FUNCTIONS: .ENABL AND .DSABL DIRECTIVES  5-7
5.3 DATA STORAGE DIRECTIVES  5-8
5.3.1 .BYTE  5-8
5.3.2 .WORD  5-9
5.3.3 ASCII Conversion of One or Two Characters  5-10
5.3.4 .ASCII  5-11
5.3.5 .ASCIz  5-12
5.3.6 .RAD50  5-12
5.4 RADIX CONTROL  5-14
5.4.1 .RADIX  5-14
5.4.2 Temporary Radix Control: †D, †O, and †B  5-14
5.5 LOCATION COUNTER CONTROL  5-15
5.5.1 .EVEN  5-15
5.5.2 .ODD  5-16
5.5.3 .BLKB and .BLKW  5-17
5.6 TERMINATING DIRECTIVES  5-17
5.6.1 .END  5-17
5.7 CONDITIONAL ASSEMBLY DIRECTIVES  5-18
5.7.1 Subconditionals  5-19
5.7.2 Immediate Conditionals  5-21
5.7.3 PAL-lIR Conditional Assembly Directives  5-22

CHAPTER 6  MACRO DIRECTIVES

6.1 MACRO DEFINITION  6-1
6.1.1 .MACRO  6-1
PART IV
OPERATING PROCEDURES

CHAPTER 7
OPERATING PROCEDURES

7.1 LOADING MAC11
7.2 COMMAND INPUT STRING

APPENDICES

APPENDIX A
A.1 ASCII CHARACTER SET
A.2 RADIX-50 CHARACTER SET

APPENDIX B
B.1 MAC11 ASSEMBLY LANGUAGE AND ASSEMBLER
B.2 SPECIAL CHARACTERS
B.3 ADDRESS MODE SYNTAX
B.3.1 Double-Operand Instructions
B.3.2 Single-Operand Instructions
B.3.3 Operate Instructions
B.3.4 Trap Instructions
B.3.5 Branch Instructions
B.3.6 Register Destination
B.3.7 Subroutine Return
B.4 ASSEMBLER DIRECTIVES

APPENDIX C
PERMANENT SYMBOL TABLE
This manual describes the PDP-11 MACRO-ll 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-ll Assembler operates under the PDP-15 DOS (Disk Operating System) Monitor in conjunction with FIREX, a multiprogramming executive running on a PDP-ll 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
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-ll assembler for the PDP-11, is specifically written for the Unichannel-15 system. Programs written for the MACRO-ll assembler will not necessarily assemble correctly with MAC11, and programs written for MAC11 will not necessarily assemble correctly with MACRO-ll.

The MAC11 assembler generates only absolute binary output.
PART 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 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 lends 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 002060 10$; CALL 20$ ;MOVE A CHARACTER
2 002064 003375 BGT 10$ ;LOOP IF GT ZERO
3 002066 001432 BFQ 19$ ;END IF ZERO
4 002070 114200 MOV -(R2),R0 ;TERMINATOR, BACK UP
5 002072 020027 CMP R0,#MT,MAX ;END OF TYPE?
    177603
6 002076 101453 BLOS 22$ ; YES
7 002100 010146 MOV RL,-(SP) ;REMEMBER READ POINTER
8 002102 016701 MOV MSBARG,RL
    002034
9 002106 005721 TST (RL)+
10 002110 010203 MOV R2,R3 ; AND WRITE POINTER
11 002112 005400 NFG R0 ;ASSUME MACRO
12 002114 026727 CMP MSBTYP,#MT,MAC ;TRUE?
    002026
    177603
13 02122 001402 BFQ 12$ ; YES, USE IT
14 02124 016700 MOV MSBCT,R0 ;GET ARG NUMBER
    002036
15 02130 010302 12$; MOV R3,R2 ;RESET WRITE POINTER
16 02132 13$; CALI 20$ ;MOVE A BYTE
17 02136 003375 BGT 13$ ;LOOP IF PTNZ
18 02140 002402 BLT 14$ ;END IF LESS THAN ZERO
19 02142 005300 DFC R0 ;ARE WE THERE YET?
20 02144 003371 BFQ R0 ; NO
21 02146 105742 14$; TSTB -(R2) ;YES, BACK UP POINTER
22 02150 012601 MOV (SP)+,RL ;RESET READ POINTER
23 02152 000742 BR 10$ ;END OF ARGUMENT
24 ; SUBSTITUTION
25 02154 010167 19$; MOV RL,MSBMRP ;END OF LINE, SAVE
```
002042'  02160 052767  BIS #LC,ME,LCFLAG ;FLAG AS MACRO
      000400  ;EXPANSION
      000010'
28  02166 000726  BR  9$
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$:  CMP R2,#LINBUF+SRCLFN ;OVERFLOW?
35  02210 101404  BLO$ 23$ ; NO
36  02212  ERROR L ;YES, FLAG ERROR
37  02220 105742  TSTB -(R2) ; AND MOVE POINTER
      23$:  BACK
38  02222 112122  MOVB (R1)+, (R2)+ ;MOVE CHAR INTO LINE
      23$:  BUFFER
39  02224  RETURN
40
41  02226  22$:  CALL ENDMAC ;CLOSE MACRO
42  02232 000167  JMP 1$  . ENDC
43
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:

\[
\begin{align*}
\text{MOV A,B} & \quad ; \text{THREE WORDS OF STORAGE} \\
\text{MOV R0,R1} & \quad ; \text{ONE WORD OF STORAGE}
\end{align*}
\]

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:

\[
\begin{align*}
\text{SYM} &= 42 \\
\text{MOV} & \ \#\text{SYM}, \text{R0}
\end{align*}
\]

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

\[
\begin{align*}
\text{R0} &= \%0 \\
\text{R1} &= \%1 \\
\text{R2} &= \%2 \\
\text{R3} &= \%3 \\
\text{R4} &= \%4 \\
\text{R5} &= \%5 \\
\text{SP} &= \%6 \quad (\text{processor stack pointer}) \\
\text{PC} &= \%7 \quad (\text{program counter})
\end{align*}
\]

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:

\[
\text{JSR} \quad \text{R5, SUBA}
\]

but is direct and fast.
2. A TRAP instruction which requires one in-line PDP-11 word:

\[
\text{TRAP } N
\]

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 \( N \)th subroutine in the table.

1.5.2 Register Increment

The operation:

\[
\text{CMPB (R0)+, (R0)+}
\]

is preferable to:

\[
\text{TST (R0)+}
\]

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.

<table>
<thead>
<tr>
<th>SIGNED</th>
<th>UNSIGNED</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGE</td>
<td>BHIS (BCC)</td>
</tr>
<tr>
<td>BLT</td>
<td>BLO</td>
</tr>
<tr>
<td>BGT</td>
<td>BHI</td>
</tr>
<tr>
<td>BLE</td>
<td>BLOS (BCS)</td>
</tr>
</tbody>
</table>

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.

MAC1 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:

<table>
<thead>
<tr>
<th>Label Field</th>
<th>Operator Field</th>
<th>Operand Field</th>
<th>Comment Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK=-10</td>
<td>REGEKP:</td>
<td></td>
<td>;REGISTER EXPRESSION</td>
</tr>
<tr>
<td>REGTST:</td>
<td>BIT</td>
<td>#MASK,VALUE</td>
<td>;3 BITS?</td>
</tr>
<tr>
<td>REGERR:</td>
<td>ERROR</td>
<td>R</td>
<td>;NO, ERROR</td>
</tr>
<tr>
<td>REGERX:</td>
<td>MOV</td>
<td>#DEFFLG:REGFLG,MODE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BIC</td>
<td>#MASK,VALUE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BR</td>
<td>ABSEXPR</td>
<td></td>
</tr>
</tbody>
</table>

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.
PART 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", "ASCIIZ", " (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:

<table>
<thead>
<tr>
<th>Character</th>
<th>Designation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>carriage return</td>
<td></td>
<td>formatting character</td>
</tr>
<tr>
<td>line feed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>form feed</td>
<td></td>
<td>source statement terminators</td>
</tr>
<tr>
<td>vertical tab</td>
<td>colon</td>
<td>label terminator</td>
</tr>
<tr>
<td></td>
<td>equal sign</td>
<td>direct assignment indicator</td>
</tr>
<tr>
<td></td>
<td>percent sign</td>
<td>register term indicator</td>
</tr>
<tr>
<td></td>
<td>tab</td>
<td>item or field terminator</td>
</tr>
</tbody>
</table>

3-1
space
#
@
()
)
,
;
<
>
+
-
*
/
&
!
"
'
^\n
item or field terminator
immediate expression indicator
defered addressing indicator
initial register indicator
terminal register indicator
operand field separator
comment field indicator
initial argument or expression indicator
terminal argument or expression indicator
arithmetic addition operator or autoincrement indicator
arithmetic subtraction operator or autodecrement indicator
arithmetic multiplication operator
arithmetic division operator
logical AND operator
logical inclusive OR operator
double ASCII character indicator
single ASCII character indicator
universal operator, argument indicator
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

<table>
<thead>
<tr>
<th>Character</th>
<th>Definition</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>one or more spaces and/or tabs</td>
<td>A space is a legal separator only for argument operands. Spaces within expressions are ignored (see Section 3.8).</td>
</tr>
<tr>
<td>,</td>
<td>comma</td>
<td>A comma is a legal separator for both expressions and the argument operands.</td>
</tr>
</tbody>
</table>

Table 3-2
Legal Delimiting Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Definition</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;...&gt;</td>
<td>paired angle brackets</td>
<td>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.</td>
</tr>
<tr>
<td>↑...\</td>
<td>Up arrow construction where the up arrow character is followed by an argument bracketed by any paired printing characters.</td>
<td>This construction is equivalent in function to the paired angle brackets and is generally used only where the argument contains angle brackets.</td>
</tr>
</tbody>
</table>

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<BC>
```
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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Explanation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>plus sign</td>
<td>+A</td>
</tr>
<tr>
<td>-</td>
<td>minus sign</td>
<td>-A</td>
</tr>
<tr>
<td>†</td>
<td>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)</td>
<td></td>
</tr>
</tbody>
</table>

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

```
-§5
†C©012
```
<table>
<thead>
<tr>
<th>Binary Operator</th>
<th>Explanation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>A+B</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>A-B</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>A*B</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>A/B</td>
</tr>
<tr>
<td>&amp;</td>
<td>logical AND</td>
<td>A&amp;B</td>
</tr>
<tr>
<td>l</td>
<td>logical inclusive OR</td>
<td>A</td>
</tr>
</tbody>
</table>

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 operand 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):

\[
\begin{align*}
X &= Y \\
Y &= Z \\
Z &= 1
\end{align*}
\]

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:

\[
\begin{align*}
\%0 \\
\%1 \\
\vdots
\end{align*}
\]

\[
\%7
\]

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:

\[
\begin{align*}
8 & \hspace{0.5cm} 000000 & R0=&0 \\
9 & \hspace{0.5cm} 000001 & R1=&1 \\
10 & \hspace{0.5cm} 000002 & R2=&2 \\
11 & \hspace{0.5cm} 000003 & R3=&3 \\
12 & \hspace{0.5cm} 000004 & R4=&4 \\
13 & \hspace{0.5cm} 000005 & R5=&5 \\
14 & \hspace{0.5cm} 000006 & R6=&6 \\
15 & \hspace{0.5cm} 000006 & SP=&6 \\
16 & \hspace{0.5cm} 000007 & PC=&7 \\
17 & \hspace{0.5cm} 000007 & R7=&7 \\
18 & \hspace{0.5cm} 000007 & R7=&7
\end{align*}
\]

;REGISTER DEFINITION
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.

<table>
<thead>
<tr>
<th>Ln.</th>
<th>Octal</th>
<th>Expansion</th>
<th>Source Code</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>SBTL SECTOR INITIALIZATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>00000</td>
<td>XCPFRG:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>00000</td>
<td>012700</td>
<td>MOV $IMPURE,R0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>00004</td>
<td>005020 1$:</td>
<td>CLR (R0)+ ;CLEAR IMPURE AREA</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>00006</td>
<td>022700</td>
<td>CMP $IMPTOP,R0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>000040</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>00012</td>
<td>101374</td>
<td>BHI 1$</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>00000</td>
<td></td>
<td></td>
<td>;PASS INITIALIZATION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>;CODE</td>
</tr>
<tr>
<td>18</td>
<td>00000</td>
<td>XCPAS:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>00000</td>
<td>012700</td>
<td>MOV $IMPPAS,R0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>00004</td>
<td>005020 1$:</td>
<td>CLR (R0)+ ;CLEAR IMPURE PART</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>00006</td>
<td>022700</td>
<td>CMP $IMPTOP,R0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>000040</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>00012</td>
<td>101374</td>
<td>BHI 1$</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>00000</td>
<td></td>
<td></td>
<td>;LINE INITIALIZATION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>;CODE</td>
</tr>
<tr>
<td>25</td>
<td>00000</td>
<td>XCTLIN:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>00020</td>
<td>012700</td>
<td>MOV $IMPLIN,R0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>00004</td>
<td>005020 1$:</td>
<td>CLR (R0)+</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>00006</td>
<td>022700</td>
<td>CMP $IMPTOP,R0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>000040</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>00012</td>
<td>101374</td>
<td>BHI 1$</td>
<td></td>
</tr>
</tbody>
</table>

3-9
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 ; . REFERES 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 1 LA 177777
```

is evaluated as

```
TAG 1 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
B = 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:
   
   \[ R0 = %0 \quad ; \text{GENERAL REGISTER 0} \]
   \[ RL = R0+1 \quad ; \text{GENERAL REGISTER 1} \]
   \[ R2 = 1+%1 \quad ; \text{GENERAL REGISTER 2} \]

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.

4-3
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
CLR @@X ;CONTENT OF LOCATION 100

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

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-,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:

\[
\text{MOV} @X, R0
\]

;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.

<table>
<thead>
<tr>
<th>Form</th>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>0n</td>
<td>Register mode</td>
</tr>
<tr>
<td>( @R ) or ( (ER) )</td>
<td>1n</td>
<td>Register deferred mode</td>
</tr>
<tr>
<td>( (ER) + )</td>
<td>2n</td>
<td>Autoincrement mode</td>
</tr>
<tr>
<td>( @(ER) + )</td>
<td>3n</td>
<td>Autoincrement deferred mode</td>
</tr>
<tr>
<td>( -(ER) )</td>
<td>4n</td>
<td>Autodecrement mode</td>
</tr>
<tr>
<td>( @-(ER) )</td>
<td>5n</td>
<td>Autodecrement deferred mode</td>
</tr>
</tbody>
</table>

where \( n \) is the register number.

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

<table>
<thead>
<tr>
<th>Form</th>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E(ER) )</td>
<td>6n</td>
<td>Index mode</td>
</tr>
<tr>
<td>( @E(ER) )</td>
<td>7n</td>
<td>Index deferred mode</td>
</tr>
<tr>
<td>( #E )</td>
<td>27</td>
<td>Immediate mode</td>
</tr>
<tr>
<td>( @#E )</td>
<td>37</td>
<td>Absolute memory reference mode</td>
</tr>
<tr>
<td>( @E )</td>
<td>67</td>
<td>Relative mode</td>
</tr>
<tr>
<td>( @E )</td>
<td>77</td>
<td>Relative deferred reference mode</td>
</tr>
</tbody>
</table>

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.
PART 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 MACll 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):

<table>
<thead>
<tr>
<th>Argument</th>
<th>Default</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEQ</td>
<td>list</td>
<td>Controls the listing of source line sequence numbers. Error flags are normally printed on the line preceding the questionable source statement.</td>
</tr>
<tr>
<td>LOC</td>
<td>list</td>
<td>Controls the listing of the location counter (this field would not normally be suppressed).</td>
</tr>
<tr>
<td>BIN</td>
<td>list</td>
<td>Controls the listing of generated binary code.</td>
</tr>
<tr>
<td>BEX</td>
<td>list</td>
<td>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.</td>
</tr>
<tr>
<td>SRC</td>
<td>list</td>
<td>Controls the listing of the source code.</td>
</tr>
<tr>
<td>COM</td>
<td>list</td>
<td>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.</td>
</tr>
<tr>
<td>MD</td>
<td>list</td>
<td>Controls listing of macro definitions and repeat range expansions.</td>
</tr>
<tr>
<td>MC</td>
<td>list</td>
<td>Controls listing of macro calls and repeat range expressions.</td>
</tr>
<tr>
<td>ME</td>
<td>no list</td>
<td>Controls listing of macro expansions.</td>
</tr>
<tr>
<td>MEB</td>
<td>no list</td>
<td>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.</td>
</tr>
</tbody>
</table>
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 .SETTL 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.
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.

| 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/DISABL FUNCTIONS       |
| 75-1 | CROSS REFERENCE HANDLERS     |
| 78-1 | LISTING STUFF               |
| 79-1 | KEYBOARD 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:

<table>
<thead>
<tr>
<th>Symbolic Argument</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDR</td>
<td>The statement .ENABL CDR causes source columns 73 and greater to be treated as comment. This accommodates sequence numbers in card columns 72-80.</td>
</tr>
<tr>
<td>LC</td>
<td>Enabling of this function causes the Assembler to accept lower case ASCII input instead of converting it to upper case.</td>
</tr>
<tr>
<td>LSB</td>
<td>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</td>
</tr>
</tbody>
</table>
.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
.ASCII
.READ50
†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 exp1,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
    LABEL: +MOV, LABEL ;THE OP-CODE FOR MOV, WHICH IS 010000,
    ;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:

<table>
<thead>
<tr>
<th>15</th>
<th>8</th>
<th>7</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

octal ASCII value of A

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 @CHRPTNT,'': ;COLOM DELIMITS LABEL FIELD.
BEQ LABEL
CMPB @CHRPTNT,'=' ;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:

<table>
<thead>
<tr>
<th>15</th>
<th>8</th>
<th>7</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>122</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

octal ASCII value of B   octal ASCII value of A
The " character is never followed by a carriage return, null, rubout, line feed or form feed. For example:

;DEVICE NAME TABLE

<table>
<thead>
<tr>
<th>DEVNAME</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>.WORD &quot;DF&quot; ;RF DISK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.WORD &quot;DK&quot; ;RR DISK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.WORD &quot;DP&quot; ;RP DISK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEVNKB</th>
</tr>
</thead>
<tbody>
<tr>
<td>.WORD &quot;KB&quot; ;TTY KEYBOARD</td>
</tr>
<tr>
<td>.WORD &quot;DT&quot; ;DECTAPE</td>
</tr>
<tr>
<td>.WORD &quot;LP&quot; ;LINE PRINTER</td>
</tr>
<tr>
<td>.WORD &quot;PR&quot; ;PAPER TAPE READER</td>
</tr>
<tr>
<td>.WORD &quot;PP&quot; ;PAPER TAPE PUNCH</td>
</tr>
<tr>
<td>.WORD &quot;CR&quot; ;CARD READER</td>
</tr>
<tr>
<td>.WORD &quot;MT&quot; ;MAGTAPE</td>
</tr>
<tr>
<td>.WORD 0 ;TABLE'S END</td>
</tr>
</tbody>
</table>

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:

can be 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:

      5-11
<table>
<thead>
<tr>
<th>Example</th>
<th>ASCII string Generated</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ASCII ;ABC;/DEF/</td>
<td>A B C D E F</td>
<td>Acceptable, but not a recommended procedure.</td>
</tr>
<tr>
<td>.ASCII /ABC;/DEF;</td>
<td>A B C</td>
<td>;DEF; is treated as a comment and ignored.</td>
</tr>
<tr>
<td>.ASCII /ABC=/DEF=</td>
<td>A B C D E F</td>
<td>Acceptable, but not recommended procedure.</td>
</tr>
<tr>
<td>.ASCII =DEF=</td>
<td></td>
<td>The assignment .ASCII=DEF is performed and a Q-error is generated upon encountering the second =.</td>
</tr>
</tbody>
</table>

5.3.5 .ASCII

The .ASCII 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 .ASCII 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: .ASCII <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

can be any printing 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:

<table>
<thead>
<tr>
<th>Character</th>
<th>Radix-50 Equivalent (octal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(space)</td>
<td>0</td>
</tr>
<tr>
<td>A-Z</td>
<td>1-32</td>
</tr>
<tr>
<td>$</td>
<td>33</td>
</tr>
<tr>
<td>.</td>
<td>34</td>
</tr>
<tr>
<td>0-9</td>
<td>36-47</td>
</tr>
</tbody>
</table>

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, .ASCIIZ, 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.

5-14
MAC11 has three unary operators to provide a single interpretation in a given radix within another radix as follows:

\[ \uparrow Dx \quad (x \text{ is treated as being in decimal radix}) \]
\[ \uparrow Ox \quad (x \text{ is treated as being in octal radix}) \]
\[ \uparrow Bx \quad (x \text{ is treated as being in binary radix}) \]

For example:

\[ \uparrow D123 \]
\[ \uparrow O 47 \]
\[ \uparrow B 00001101 \]
\[ \uparrow O(A+3) \]

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 MAC11, allowing a temporary radix change from octal to decimal by specifying a decimal radix number with a "decimal point". For example:

\[ 100. \quad (144(8)) \]
\[ 1376. \quad (2540(8)) \]
\[ 128. \quad (200(8)) \]

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:

```
.EVEN /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
    ;MOV R(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
    ;
    ;
    ;CLR B -(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      ;NEXT GROUP MUST STAY TOGETHER
2 000002  SYMBOL: .BLKW 2  ;SYMBOL ACCUMULATOR
3 000006  MODE: .BLKW 1   ;FLAG BITS
4 000006  FLAGS: .BLKW 1  ;SYMBOL/EXPRESSIONS TYPE
5 000007  SECTOR: .BLKW 1 ;EXPRESSION VALUE
6 000010  VALUE: .BLKW 1  ;END OF GROUPED DATA
7 000120  RELLVI: .BLKW 1
8 000128  RELLVI: .BLKW 2
```

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, MACLL 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:

 Получить начало условного блока

```
    .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:

<table>
<thead>
<tr>
<th>Conditions</th>
<th>POSITIVE</th>
<th>COMPLEMENT</th>
<th>ARGUMENTS</th>
<th>ASSEMBLE BLOCK IF</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>NE</td>
<td>expression</td>
<td>expression=0 (or =0)</td>
<td></td>
</tr>
<tr>
<td>GT</td>
<td>LE</td>
<td>expression</td>
<td>expression&gt;0 (or &lt;0)</td>
<td></td>
</tr>
<tr>
<td>LT</td>
<td>GE</td>
<td>expression</td>
<td>expression&lt;0 (or &gt;0)</td>
<td></td>
</tr>
<tr>
<td>DF</td>
<td>NDF</td>
<td>symbolic argument</td>
<td>symbol is defined (or undefined)</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>NB</td>
<td>macro-type argument</td>
<td>argument is a blank (or not blank)</td>
<td></td>
</tr>
</tbody>
</table>
IDN | DIF | two macro-type arguments identical arguments separated (or different) by a comma
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
I 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:

<table>
<thead>
<tr>
<th>Subconditional</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>.IFF</td>
<td>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 false.</td>
</tr>
<tr>
<td>.IFT</td>
<td>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.</td>
</tr>
<tr>
<td>.IFTF</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 DEFINED.
 .IFT
   ; ASSEMBLE THE FOLLOWING CODE ONLY IF SYM IS DEFINED.
 .IFTF
   ; ASSEMBLE THE FOLLOWING CODE UNCONDITIONALLY.

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

5-20
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
```

5-21
5.7.3 PAL-llR Conditional Assembly Directives

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

<table>
<thead>
<tr>
<th>Directive</th>
<th>Arguments</th>
<th>Assemble Block if</th>
</tr>
</thead>
<tbody>
<tr>
<td>.IFZ or .IFEQ</td>
<td>expression</td>
<td>expression=0</td>
</tr>
<tr>
<td>.IFNZ or .IFNE</td>
<td>expression</td>
<td>expression=0</td>
</tr>
<tr>
<td>.IFL or .IFLT</td>
<td>expression</td>
<td>expression&lt;0</td>
</tr>
<tr>
<td>.IFG or .IFGT</td>
<td>expression</td>
<td>expression&gt;0</td>
</tr>
<tr>
<td>.IFLE</td>
<td>expression</td>
<td>expression&lt; or =0</td>
</tr>
<tr>
<td>.IFGE</td>
<td>expression</td>
<td>expression&gt; or =0</td>
</tr>
<tr>
<td>.IFDF</td>
<td>logical expression</td>
<td>expression is true (defined)</td>
</tr>
<tr>
<td>.IFNDF</td>
<td>logical expression</td>
<td>expression is false (undefined)</td>
</tr>
</tbody>
</table>

The rules governing the usage of these directives are now the same as for the MACRO-ll 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-llR).
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 MESSAGE  ;TYPE A MESSAGE
JSR R5,TYPMSG
.WORD MESSAGE
.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 expansion.
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:

\[ .MACRO \ A,A,B,C \]

\[ \text{REN} \quad \text{ALPHA, BETA, } <\text{Cl, C2}> \]

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:

\[ \text{REN} \ <\text{MOV X,Y}>\#44, \text{WEV} \]

This call would cause the entire statement:

\[ \text{MOV X,Y} \]

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:

\[ \text{REN} \ \^/\text{MOV X,Y}/, \#44, \text{WEV} \]

which is equivalent to

\[ \text{REN} \ <\text{MOV X,Y}>, \#44, \text{WEV} \]

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

\[ \text{REN} \ \#44, \text{WEV} \ ^/\text{MOV X,Y}/ \]

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.

\[ .MACRO \ LEV1 \ DUM1, DUM2 \]

\[ \text{LEVEL2 \ DUM1} \]

\[ \text{LEVEL2 \ DUM2} \]

\[ .ENDM \]
.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
\ASCII /SYM/
\ENDM
\MACRO OUT ARG IDT 005A\ARG
\ENDM

\OUT \ID

;ASSUME THAT THE SYMBOL ID TAKES
;ON A UNIQUE TWO DIGIT VALUE FOR
;EACH POSSIBLE CONDITIONAL ASSEMBLY
;OF THE PROGRAM
;WHERE 005A IS THE UPDATE
;VERSION OF THE PROGRAM
;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 .IFNEB can be used within the macro to detect unnecessary arguments.

A macro can be defined with no arguments.

6.3.5 Automatically Created Symbols

MAC11 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
BEC 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: .ASCIIZ "C/
    .WORD '"A""B
    .ENDM
```

When this macro is called:

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

it expands as follows:

```
XY: .ASCIIZ /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 .ASCIIZ 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:

label 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 MAC111 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
    .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.
.TITLE IRPTST
.LIST MD,MC,ME

000000 R0=0 00
000001 R1=0 01
000002 R2=0 02
000003 R3=0 03
000004 R4=0 04
000005 R5=0 05
000006 R6=0 06
000007 R7=0 07
000008 SP=0 07
000007 PC=0 07
177776 PSW= 0177776
177570 SWR= 0177570

3 000000 012/00 MOV #TABLE,R0
000050

4 .IRP X,<A,B,C,D,E,F>
5
6 MOV X,(R0)+
7
8 .ENDM
9

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 .IRPC X,ABCDEF
13
14 .ASCII /X/
15
16 .ENDM
17

00034 101 .ASCII /A/
00035 102 .ASCII /B/
00036 103 .ASCII /C/
00037 104 .ASCII /D/

6-13
Figure 6-1
.INR 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:

```assembly
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:

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

6-14
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.
PART IV
OPERATING PROCEDURES

This part of the manual describes the operation of the MACII 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 VIA
>

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=FILENAME

where:

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

FILENAME 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 input 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
OBJECT CODE HANDLERS

1
2
3 012026 ENDP;
4 012026 CALL SETMAX
 012026 JSR PC,SETMAC
 34767 174248
5 012062 TST PASS
 05767 iPASS ONE?
 000007';
6 012036 BNE ENDP2
 01142 ;BRANCH IF PASS 2
7 012040 4
8 012040 TST OBJLNK
 05767 ;PASS ONE, ANY OBJECT?
 01416';
9 012044 BEQ 30S
 01517 ; NO
 000001
10 012046 MOV #BLK101,BLKTYP
 012767 ;SET BLOCK TYPE 1
 000002
11 000542'
11 12054 CALL OBJJIN
 1054 CALL OBJJIN
 04767 001542
12 12060 JSR PC,OBJJIN
 012701 iINIT THE POINTERS
 000050';
13 12064 MOV #PRGTTT1,R1
 016702 ;SET "FROM" INDEX
 000549'
14 12070 MOV RLDPNT,R2
 12070 MOV RLDPNT,R2
 04767 000566'
15 12074 CALL GSDDMP
 05846 ;OUTPUT GSD BLOCK
 12074 CALL GSDDMP
 05946 ';
16 12076 MOV (SP),ROLUD
 12076 MOV (SP),ROLUD
 012687 000000'
17 10S1
17 12102 NEXT SECROL
 12102 NEXT SECROL
 012700 000010
18 12116 MOV #SECROL,R3
 12116 MOV #SECROL,R3
 04767 000540
19 12112 BEQ 20S
 12112 BEQ 20S
 01450 iBRANCH IF THROUGH
 000050'
19 12114 MOV ROLUD,=(SP)
 12114 MOV ROLUD,=(SP)
 06746 000006'
21 12124 SAVE MARKER
 12124 SAVE MARKER
 011105 000000
22 12126 MOV (R1),R5
 12126 MOV (R1),R5
 042705 00377
23 12132 ISAVE SECTOR
 12132 ISAVE SECTOR
 080305 00377
24 12134 BIC #377,R5
 12134 BIC #377,R5
 042711 177737
25 177737 ;AND PLACE IN RIGHT
25 12140 BIC =1=<RELFLG>,(R1)
 12140 BIC =1=<RELFLG>,(R1)
 092721 ;CLEAR ALL BUT REL RIT
 000410
26 12144 BIS #<GSDT01>,<DEFFLG>,(R1) SET TO TYPE 1, DEFINED
 12144 BIS #<GSDT01>,<DEFFLG>,(R1) SET TO TYPE 1, DEFINED
 010521 000010
27 12146 MOV R5,(R1)+ ;ASSUME ABS
 12146 MOV R5,(R1)+ ;ASSUME ABS
 031401 11S
28 12150 BEQ OOPS!
 12150 BEQ OOPS!
 011141 7-3
28 12150 MOV (R1),=(R1) ;REL, SET MAX

Figure 7-1
Assembly Listing
Figure 7-1 (Cont.)
Assembly Listing
## APPENDIX A

### MACII CHARACTER SET

#### A.1 ASCII CHARACTER SET

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

A-1
<table>
<thead>
<tr>
<th>EVEN PARITY BIT</th>
<th>7-BIT OCTAL CODE</th>
<th>7-BIT CHARACTER</th>
<th>REMARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>030</td>
<td>CAN</td>
<td>CANCEL (CANCEL). CONTROL/X.</td>
</tr>
<tr>
<td>1</td>
<td>031</td>
<td>EM</td>
<td>END OF MEDIUM. CONTROL/Y.</td>
</tr>
<tr>
<td>1</td>
<td>032</td>
<td>SUB</td>
<td>SUBSTITUTE. CONTROL/Z.</td>
</tr>
<tr>
<td>1</td>
<td>033</td>
<td>ESC</td>
<td>ESCAPE. CONTROL/SHIFT/K.</td>
</tr>
<tr>
<td>1</td>
<td>034</td>
<td>FS</td>
<td>FILE SEPARATOR. CONTROL/SHIFT/L.</td>
</tr>
<tr>
<td>0</td>
<td>035</td>
<td>GS</td>
<td>GROUP SEPARATOR. CONTROL/SHIFT/M.</td>
</tr>
<tr>
<td>0</td>
<td>036</td>
<td>RS</td>
<td>RECORD SEPARATOR. CONTROL/SHIFT/N.</td>
</tr>
<tr>
<td>1</td>
<td>037</td>
<td>US</td>
<td>UNIT SEPARATOR. CONTROL/SHIFT/O.</td>
</tr>
<tr>
<td>1</td>
<td>040</td>
<td>SP</td>
<td>SPACE.</td>
</tr>
<tr>
<td>0</td>
<td>041</td>
<td>&quot;</td>
<td>ACCENT ACUTE OR APOSTROPE.</td>
</tr>
<tr>
<td>1</td>
<td>042</td>
<td>#</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>043</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>044</td>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>045</td>
<td>'</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>046</td>
<td>)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>047</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>050</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>051</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>052</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>053</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>054</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>055</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>056</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>057</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>060</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>061</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>062</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>063</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>064</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>065</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>066</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>067</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>070</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>071</td>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>072</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>073</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>074</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>075</td>
<td>@</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>076</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>077</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>102</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>103</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>104</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>105</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>106</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>107</td>
<td>J</td>
<td></td>
</tr>
<tr>
<td>EVEN PARITY BIT</td>
<td>7-BIT OCTAL CODE</td>
<td>CHARACTER</td>
<td>REMARKS</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------</td>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>0</td>
<td>113</td>
<td>K</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>114</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>115</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>116</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>117</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>120</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>121</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>122</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>123</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>124</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>125</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>126</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>127</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>130</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>131</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>132</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>133</td>
<td>[</td>
<td>SHIFT/K.</td>
</tr>
<tr>
<td>0</td>
<td>134</td>
<td>\</td>
<td>SHIFT/L.</td>
</tr>
<tr>
<td>1</td>
<td>135</td>
<td>]</td>
<td>SHIFT/M.</td>
</tr>
<tr>
<td>1</td>
<td>136</td>
<td>+</td>
<td>ACCENT GRAVE.</td>
</tr>
<tr>
<td>1</td>
<td>137</td>
<td>'</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>141</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>142</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>143</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>144</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>145</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>146</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>147</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>150</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>151</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>152</td>
<td>j</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>153</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>154</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>155</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>156</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>157</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>160</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>161</td>
<td>q</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>162</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>163</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>164</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>165</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>166</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>167</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>170</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>171</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>172</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>173</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>174</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>175</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>176</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>177</td>
<td>DEL</td>
<td>DELETE, RUBOUT.</td>
</tr>
</tbody>
</table>

*This code generated by altmode.*
*This code generated by prefix key (if present).*
A.2 RADIX-50 CHARACTER SET

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII Octal Equivalent</th>
<th>Radix-50 Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>A-Z</td>
<td>101 - 132</td>
<td>1 - 32</td>
</tr>
<tr>
<td>$</td>
<td>44</td>
<td>33</td>
</tr>
<tr>
<td>*</td>
<td>56</td>
<td>34</td>
</tr>
<tr>
<td>unused</td>
<td>60 = 71</td>
<td>36 = 47</td>
</tr>
</tbody>
</table>

The maximum Radix-50 value is, thus,

\[47\times50(2) + 47\times50 + 47 = 174777\]

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):

\[
\begin{align*}
X &= 113000 \\
2 &= 002400 \\
B &= 000002 \\
X2B &= 115402
\end{align*}
\]
<table>
<thead>
<tr>
<th>Single Char. or First Char.</th>
<th>Second Character</th>
<th>Third Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>003100</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>006200</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>011300</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>014400</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>017500</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>022600</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>025700</td>
<td>G</td>
</tr>
<tr>
<td>H</td>
<td>031000</td>
<td>H</td>
</tr>
<tr>
<td>I</td>
<td>034100</td>
<td>I</td>
</tr>
<tr>
<td>J</td>
<td>037200</td>
<td>J</td>
</tr>
<tr>
<td>K</td>
<td>042300</td>
<td>K</td>
</tr>
<tr>
<td>L</td>
<td>045400</td>
<td>L</td>
</tr>
<tr>
<td>M</td>
<td>050500</td>
<td>M</td>
</tr>
<tr>
<td>N</td>
<td>053600</td>
<td>N</td>
</tr>
<tr>
<td>O</td>
<td>056700</td>
<td>O</td>
</tr>
<tr>
<td>P</td>
<td>062000</td>
<td>P</td>
</tr>
<tr>
<td>Q</td>
<td>065100</td>
<td>Q</td>
</tr>
<tr>
<td>R</td>
<td>070200</td>
<td>R</td>
</tr>
<tr>
<td>S</td>
<td>073300</td>
<td>S</td>
</tr>
<tr>
<td>T</td>
<td>076400</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>101500</td>
<td>U</td>
</tr>
<tr>
<td>V</td>
<td>104600</td>
<td>V</td>
</tr>
<tr>
<td>W</td>
<td>107700</td>
<td>W</td>
</tr>
<tr>
<td>X</td>
<td>113000</td>
<td>X</td>
</tr>
<tr>
<td>Y</td>
<td>116100</td>
<td>Y</td>
</tr>
<tr>
<td>Z</td>
<td>121200</td>
<td>Z</td>
</tr>
<tr>
<td>$</td>
<td>124300</td>
<td>$</td>
</tr>
<tr>
<td>0</td>
<td>127400</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>132500</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>135600</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>140700</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>144800</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>147900</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>152200</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>155300</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>160400</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>163500</td>
<td>9</td>
</tr>
</tbody>
</table>

A-5
# APPENDIX B

## MAC11 ASSEMBLY LANGUAGE AND ASSEMBLER

### B.1 SPECIAL CHARACTERS

<table>
<thead>
<tr>
<th>Character</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>form feed</td>
<td>Source line terminator</td>
</tr>
<tr>
<td>line feed</td>
<td>Source line terminator</td>
</tr>
<tr>
<td>carriage return</td>
<td>Formatting character</td>
</tr>
<tr>
<td>vertical tab</td>
<td>Source line terminator</td>
</tr>
<tr>
<td>( : )</td>
<td>Label terminator</td>
</tr>
<tr>
<td>( = )</td>
<td>Direct assignment indicator</td>
</tr>
<tr>
<td>( % )</td>
<td>Register term indicator</td>
</tr>
<tr>
<td>tab</td>
<td>Item terminator</td>
</tr>
<tr>
<td>space</td>
<td>Field terminator</td>
</tr>
<tr>
<td>( # )</td>
<td>Immediate expression indicator</td>
</tr>
<tr>
<td>( @ )</td>
<td>Deferred addressing indicator</td>
</tr>
<tr>
<td>( ( )</td>
<td>Initial register indicator</td>
</tr>
<tr>
<td>( ) )</td>
<td>Terminal register indicator</td>
</tr>
<tr>
<td>( , (comma) )</td>
<td>Operand field separator</td>
</tr>
<tr>
<td>( ; )</td>
<td>Comment field indicator</td>
</tr>
<tr>
<td>( + )</td>
<td>Arithmetic addition operator or autoincrement indicator</td>
</tr>
<tr>
<td>( - )</td>
<td>Arithmetic subtraction operator or autoincrement indicator</td>
</tr>
<tr>
<td>( * )</td>
<td>Arithmetic multiplication operator</td>
</tr>
<tr>
<td>( / )</td>
<td>Arithmetic division operator</td>
</tr>
<tr>
<td>( &amp; )</td>
<td>Logical AND operator</td>
</tr>
<tr>
<td>( ! )</td>
<td>Logical OR operator</td>
</tr>
<tr>
<td>( # )</td>
<td>Double ASCII character indicator</td>
</tr>
</tbody>
</table>

B-1
<table>
<thead>
<tr>
<th>Format</th>
<th>Address Mode Name</th>
<th>Address Mode Number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Register</td>
<td>0n</td>
<td>Register R contains the operand. R is a register expression.</td>
</tr>
<tr>
<td>@R or (ER)</td>
<td>Deferred Register</td>
<td>1n</td>
<td>Register R contains the operand address.</td>
</tr>
<tr>
<td>(ER)+</td>
<td>Autoincrement</td>
<td>2n</td>
<td>The contents of the register specified by ER are incremented after being used as the address of the operand.</td>
</tr>
<tr>
<td>@(ER)+</td>
<td>Deferred Autoincrement</td>
<td>3n</td>
<td>ER contains the pointer to the address of the operand. ER is incremented after use.</td>
</tr>
<tr>
<td>-(ER)</td>
<td>Autodecrement</td>
<td>4n</td>
<td>The contents of register ER are decremented before being used as the address of the operand.</td>
</tr>
<tr>
<td>@-(ER)</td>
<td>Deferred Autodecrement</td>
<td>5n</td>
<td>The contents of register ER are decremented before being used as the pointer to the address of the operand.</td>
</tr>
<tr>
<td>E(ER)</td>
<td>Index</td>
<td>6n</td>
<td>E plus the contents of the register specified, ER, is the address of the operand.</td>
</tr>
<tr>
<td>@E(ER)</td>
<td>Deferred Index</td>
<td>7n</td>
<td>E added to ER gives the pointer to the address of the operand.</td>
</tr>
</tbody>
</table>

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.
#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:

---

B-3
* 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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>1</code></td>
<td>Inclusive OR</td>
</tr>
<tr>
<td><code>⊕</code></td>
<td>Exclusive OR</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>AND</td>
</tr>
<tr>
<td><code>¬</code></td>
<td>(used over a symbol) NOT (i.e., 1's complement)</td>
</tr>
</tbody>
</table>

### B.3.1 Double-Operand Instructions

Instruction type format: Op A, A

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Mnemonic</th>
<th>Stands for</th>
<th>Operation</th>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>01SSDD</td>
<td>MOV</td>
<td>MOVe</td>
<td>(SE) (DE)</td>
<td>* * 0 -</td>
</tr>
<tr>
<td>11SSDD</td>
<td>MOVB</td>
<td>MOVe Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02SSDD</td>
<td>CMP</td>
<td>CoMPare</td>
<td>(SE)¬(DE)</td>
<td>* * * *</td>
</tr>
<tr>
<td>12SSDD</td>
<td>CMPB</td>
<td>CoMPare Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>03SSDD</td>
<td>BIT</td>
<td>BIT Test</td>
<td>(SE)&amp;(DE)</td>
<td>* * 0 -</td>
</tr>
<tr>
<td>13SSDD</td>
<td>BITB</td>
<td>BIT Test Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04SSDD</td>
<td>BIC</td>
<td>BIT Clear</td>
<td>(SE)&amp;(DE)→DE</td>
<td>* * 0 -</td>
</tr>
<tr>
<td>14SSDD</td>
<td>BICB</td>
<td>BIT Clear Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05SSDD</td>
<td>BI&quot;</td>
<td>BIT Set</td>
<td>(SE)↓(DE)→DE</td>
<td>* * 0 -</td>
</tr>
<tr>
<td>15SSDD</td>
<td>BISB</td>
<td>BIT Set Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>06SSDD</td>
<td>ADD</td>
<td>ADD</td>
<td>(SE)+(DE)→DE</td>
<td>* * * *</td>
</tr>
<tr>
<td>16SSDD</td>
<td>SUB</td>
<td>SUBtract</td>
<td>(DE) - (SE) + E</td>
<td>* * * *</td>
</tr>
</tbody>
</table>

### B.3.2 Single-Operand Instructions

Instruction-type format: Op A

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Mnemonic</th>
<th>Stands for</th>
<th>Operation</th>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0050DD</td>
<td>CLR</td>
<td>CClear</td>
<td>0 DE</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>1050DD</td>
<td>CLRB</td>
<td>CClear Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0051DD</td>
<td>COM</td>
<td>COMplement</td>
<td>(DE) DE</td>
<td>* * 0 1</td>
</tr>
<tr>
<td>1051DD</td>
<td>COMB</td>
<td>COMplement Byte</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B-4
0052DD INC INCrement (DE)+1 DE * * * -
1052DD INCB INCrement Byte
0053DD DEC DECrement (DE)-1 DE * * * -
1053DD DECB DECrement Byte
0054DD NEG NEGate (DE)+1 DE * * * *
1054DD NEGB NEGate Byte
0055DD ADC Add Carry (DE)+(C) DE * * * *
1055DD ADCB Add Carry Byte
0056DD SBC Subtract Carry (DE)-(C) DE * * * *
1056DD SBCB Subtract Carry Byte
0057DD TST TeST (DE)-0 DE * * 0 0
1057DD TSTB TeST Byte
0060DD ROR ROtate Right
1060DD RORB ROtate Right Byte even or odd byte * * * *
0061DD ROL ROtate Left
1061DD ROLB ROtate Left Byte even or odd byte * * * *
0062DD ASR Arithmeitic Shift Right
1062DD ASRB Arithmeitic Shift Right Byte even or odd byte * * * *
0063DD ASL Arithmeitic Shift Left
1063DD ASLB Arithmeitic Shift Left Byte even or odd 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
<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Mnemonic</th>
<th>Stands for</th>
<th>Operation</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>HALT</td>
<td>HALT</td>
<td>The computer stops all functions.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000001</td>
<td>WAIT</td>
<td>WAIT</td>
<td>The computer stops and waits for an interrupt.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000002</td>
<td>RTI</td>
<td>ReTurn from Interrupt</td>
<td>The PC and PS are popped off the SP stack:</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>((SP))→PC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(SP)+2→SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>((SP))→PS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(SP)+2→SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000005</td>
<td>RESET</td>
<td>RESET</td>
<td>Returns all I/O devices to power-on status.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000241</td>
<td>CLC</td>
<td>CLeaR Carry bit</td>
<td>0→C</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>000261</td>
<td>SEC</td>
<td>SeT Carry bit</td>
<td>1→C</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>000242</td>
<td>CLV</td>
<td>CLeaR oVerflow bit</td>
<td>0→V</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000262</td>
<td>SEV</td>
<td>SeT oVerflow bit</td>
<td>1→V</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000244</td>
<td>CLZ</td>
<td>CLeaR Zero bit</td>
<td>0→Z</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000264</td>
<td>SEZ</td>
<td>SeT Zero bit</td>
<td>1→Z</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000250</td>
<td>CLN</td>
<td>CLeaR Negative bit</td>
<td>0→N</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000270</td>
<td>SEN</td>
<td>SeT Negative bit</td>
<td>1→N</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000243</td>
<td>CVC</td>
<td>Clear oVerflow and Carry bits</td>
<td>0→V</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>000254</td>
<td>CNZ</td>
<td>Clear Negative and Zero bits</td>
<td>0→N</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
### B.3.4 Trap Instructions

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

*OP (only)*

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Mnemonic</th>
<th>Stands for</th>
<th>Operation</th>
<th>Status Word Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>000003</td>
<td>BPT</td>
<td>BreakPoint Trap</td>
<td>Trap to location 14. This is used to call ODT.</td>
<td>N Z V C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*000004</td>
<td>IOT</td>
<td>Input/Output Trap</td>
<td>Trap to location 20. This is used to call IOX.</td>
<td></td>
</tr>
<tr>
<td>104000-</td>
<td>EMT</td>
<td>EMulator Trap</td>
<td>Trap to location 30. This is used to call system programs.</td>
<td></td>
</tr>
<tr>
<td>104377</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>104400-</td>
<td>TRAP</td>
<td>TRAP</td>
<td>Trap to location 34. This is used to call any routine desired by the programmer.</td>
<td></td>
</tr>
<tr>
<td>104777</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B.3.5 Branch Instructions

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

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Mnemonic</th>
<th>Stands for</th>
<th>Condition to be met if branch is to occur</th>
</tr>
</thead>
<tbody>
<tr>
<td>0004XX</td>
<td>BR</td>
<td>BRanch always</td>
<td></td>
</tr>
<tr>
<td>0010XX</td>
<td>BNE</td>
<td>Branch if Not Equal (to zero)</td>
<td>Z=0</td>
</tr>
<tr>
<td>0014XX</td>
<td>BEQ</td>
<td>Branch if Equal (to zero)</td>
<td>Z=1</td>
</tr>
<tr>
<td>0020XX</td>
<td>BGE</td>
<td>Branch if Greater than or Equal (to zero)</td>
<td>N (\uparrow) V=0</td>
</tr>
<tr>
<td>0024XX</td>
<td>BLT</td>
<td>Branch if Less than (zero)</td>
<td>N (\uparrow) V=1</td>
</tr>
<tr>
<td>0030XX</td>
<td>BGT</td>
<td>Branch if Greater than (zero)</td>
<td>Z! (N (\uparrow) V)=0</td>
</tr>
<tr>
<td>0034XX</td>
<td>BLE</td>
<td>Branch if Less than or equal (to zero)</td>
<td>Z! (\uparrow) (N (\uparrow) V)=1</td>
</tr>
<tr>
<td>1000XX</td>
<td>BPL</td>
<td>Branch if PLus</td>
<td>N=0</td>
</tr>
<tr>
<td>1004XX</td>
<td>BMI</td>
<td>Branch if MINus</td>
<td>N=1</td>
</tr>
<tr>
<td>1010XX</td>
<td>BHI</td>
<td>Branch if Higher</td>
<td>C (\uparrow) Z=0</td>
</tr>
<tr>
<td>1014XX</td>
<td>BLOS</td>
<td>Branch if Lower or Same</td>
<td>C (\uparrow) Z=1</td>
</tr>
<tr>
<td>1020XX</td>
<td>BVC</td>
<td>Branch if oVerflow Clear</td>
<td>V=0</td>
</tr>
<tr>
<td>1024XX</td>
<td>BVS</td>
<td>Branch if oVerflow Set</td>
<td>V=1</td>
</tr>
<tr>
<td>1030XX</td>
<td>BCC (or BHIS)</td>
<td>Branch if Carry Clear (or Branch if Higher or Same)</td>
<td>C=0</td>
</tr>
<tr>
<td>1034XX</td>
<td>BCS (or BLOS)</td>
<td>Branch if Carry Set (or Branch if Lower)</td>
<td>C=1</td>
</tr>
</tbody>
</table>
### B.3.6 Register Destination

**Instruction type format:** Op ERA

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Mnemonic</th>
<th>Stands for</th>
<th>Status Word</th>
<th>Condition Codes</th>
<th>Operation</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>004RDA</td>
<td>JSR</td>
<td>Jump to SubRoutine</td>
<td>Push register on the SP stack, put the PC in the register.</td>
<td>DE TEMP (TEMP= temporary storage register internal to processor.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(SP)-2 SP (REG) (SP) (PC) REG (TEMP) PC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Mnemonic</th>
<th>Stands for</th>
<th>Operation</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>074RDA</td>
<td>XOR</td>
<td>eXclusive OR</td>
<td>(R) l DE DE * * 0 -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### B.3.7 Subroutine Return

**Instruction type format:** Op ER

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Mnemonic</th>
<th>Stands for</th>
<th>Operation</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>00020R</td>
<td>RTS</td>
<td>ReTurn from Subroutine</td>
<td>Put register in PC and pop old contents from SP stack into register</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Operation</td>
<td>Described in Manual Section</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>-----------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>'</td>
<td>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.</td>
<td>5.3.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td>A double-quote character followed by two ASCII characters generates a word containing the 7-bit ASCII representation of the two characters.</td>
<td>5.3.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>†Bn</td>
<td>Temporary radix control; causes the number n to be treated as a binary number.</td>
<td>5.4.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>†Cn</td>
<td>Creates a word containing the one's complement of n.</td>
<td>5.6.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>†Dn</td>
<td>Temporary radix control; causes the number n to be treated as a decimal number.</td>
<td>5.4.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>†On</td>
<td>Temporary radix control; causes the number n to be treated as an octal number.</td>
<td>5.4.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.ASCII string</td>
<td>Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters) one character per byte.</td>
<td>5.3.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.ASCIZ string</td>
<td>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.</td>
<td>5.3.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.BLKB exp</td>
<td>Reserves a block of storage space exp bytes long.</td>
<td>5.5.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORM</td>
<td>Operation</td>
<td>Described in Manual Section</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>-----------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>.BLKW exp</code></td>
<td>Reserves a block of storage space exp words long.</td>
<td>5.5.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>.BYTE exp1,exp2,...</code></td>
<td>Generates successive bytes of data containing the octal equivalent of the expression(s) specified.</td>
<td>5.3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>.DSABL arg</code></td>
<td>Disables the assembler function specified by the argument.</td>
<td>5.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>.ENABL arg</code></td>
<td>Provides the assembler function specified by the argument.</td>
<td>5.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>.END</code></td>
<td>Indicates the physical end of source program. An optional argument specifies the transfer address.</td>
<td>5.7.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>.END exp</code></td>
<td>Indicates the end of a conditional block.</td>
<td>5.11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>.ENDC</code></td>
<td>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.</td>
<td>6.1.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>.ENDM</code></td>
<td>Causes a text string to be output to the command device containing the optional expression specified and the indicated text string.</td>
<td>6.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>.ERROR exp,string</code></td>
<td>Ensures that the assembly location counter contains an even address by adding 1 if it is odd.</td>
<td>5.5.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>.EVEN</code></td>
<td>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.</td>
<td>5.11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>.IF cond,arg1,arg2,...</code></td>
<td>Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested false.</td>
<td>5.11.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Operation</td>
<td>Described in Manual Section</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------------------------</td>
<td>------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IPT</td>
<td>Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested true.</td>
<td>5.11.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IPTF</td>
<td>Appears only within a conditional block and indicates the beginning of a section of code to be unconditionally assembled.</td>
<td>5.11.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IIF cond, arg, statement</td>
<td>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.</td>
<td>5.11.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IRP sym,&lt;arg1, arg2,...&gt;</td>
<td>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).</td>
<td>6.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IRPC sym, string</td>
<td>Indicates the beginning of an indefinite repeat block in which the symbol specified takes on the value of successive characters in the character string.</td>
<td>6.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.LIST</td>
<td>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.</td>
<td>5.1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.LIST arg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.MACRO sym, arg1, arg2,...</td>
<td>Indicates the start of a macro named sym containing the dummy arguments specified.</td>
<td>6.1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.MEXIT</td>
<td>Causes an exit from the current macro or indefinite repeat block.</td>
<td>6.1.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NARG symbol</td>
<td>Appears only within a macro definition and equates the specified symbol to the number of characters in the string (enclosed in delimiting characters).</td>
<td>6.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Operation</td>
<td>Described in Manual Section</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------------------------------------</td>
<td>-----------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NCHR sym,string</td>
<td>Can appear anywhere in a source program; equates the symbol specified to the number of characters in the string (enclosed in delimiting characters).</td>
<td>6.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NLIST</td>
<td>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.</td>
<td>5.1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NLIST arg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NTYPE sym,arg</td>
<td>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.</td>
<td>6.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.ODD</td>
<td>Ensures that the assembly location counter contains an odd address by adding 1 if it is even.</td>
<td>5.5.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.PAGE</td>
<td>Causes the assembly listing to skip to the top of the next page.</td>
<td>5.1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.PRINT exp,string</td>
<td>Causes a text string to be output to the command device containing the optional expression specified and the indicated text string.</td>
<td>6.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.RADIX n</td>
<td>Alters the current program radix to n, where n can be 2, 4, 8, or 10.</td>
<td>5.4.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.RAD50 string</td>
<td>Generates a block of data containing the Radix-50 equivalent of the character string (enclosed in delimiting characters).</td>
<td>5.3.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.REPT exp</td>
<td>Begins a repeat block. Causes the section of code up to the next .ENDM or .ENDR to be repeated exp times.</td>
<td>6.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.SBTTL string</td>
<td>Causes the string to be printed as part of the assembly listing page header. The string part of each .SBTTL directive is collected into</td>
<td>5.1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Form</td>
<td>Operation</td>
<td>Described in Manual Section</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>-----------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a table of contents at the beginning of the assembly listing.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.TITLE string</td>
<td>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.</td>
<td>5.1.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.WORD expr,expr2,...</td>
<td>Generates successive words of data containing the octal equivalent of the expression(s) specified.</td>
<td>5.3.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX C
PERMANENT SYMBOL TABLE

PST PERMANENT SYMBOL TABLE MACRO V004A PAGE 1

.TITLE PST PERMANENT SYMBOL TABLE
COPYRIGHT 1972 DIGITAL EQUIPMENT CORPORATION

000020 DR1= 200 ;DESTRUCTIVE REFERENCE IN FIRST
000100 DR2= 100 ;DESTRUCTIVE REFERENCE IN SECOND

000020 DFLG E= 020 ;DIRECTIVE REQUIRES EVEN LOCATION
000010 DFLGBM= 010 ;DIRECTIVE USES BYTE MODE
000004 DFLCND= 004 ;CONDITIONAL DIRECTIVE
000002 DFLMAC= 002 ;MACRO DIRECTIVE

.IIF DF X45, XFLTG= 0
.IIF DF XMACRO, XSMCAL= 0

.MACRO OPCDEF NAME, CLASS, VALUE, FLAGS, COND
..IF NB <COND>
..IF DF COND
..MEXIT
..ENDC
..ENDC
..RADS50 \NAME/.
..BYTE FLAGS+0
..GLOBAL OPCLI'CLASS
..BYTE 200+OPCLI'CLASS
..WORD VALUE
..ENDM

.MACRO DIRDEF NAME, FLAGS, COND
..IF NB <COND>
..IF DF COND
..MEXIT
..ENDC
..ENDC
..GLOBAL NAME
..RADS50 \NAME/.
..BYTE FLAGS+0
..BYTE 0
..WORD NAME
..ENDM

000000 PSTBAS: ;BASE
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>000020</td>
<td>OPCDEF</td>
<td>$ADC$</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>000030</td>
<td>OPCDEF</td>
<td>$ADCB$</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>000040</td>
<td>OPCDEF</td>
<td>$ADD$</td>
<td>02</td>
</tr>
<tr>
<td>4</td>
<td>0000110</td>
<td>OPCDEF</td>
<td>$ASL$</td>
<td>04</td>
</tr>
<tr>
<td>5</td>
<td>0000120</td>
<td>OPCDEF</td>
<td>$ASLB$</td>
<td>01</td>
</tr>
<tr>
<td>6</td>
<td>0000130</td>
<td>OPCDEF</td>
<td>$ASR$</td>
<td>01</td>
</tr>
<tr>
<td>7</td>
<td>0000140</td>
<td>OPCDEF</td>
<td>$ASRB$</td>
<td>01</td>
</tr>
<tr>
<td>8</td>
<td>0000150</td>
<td>OPCDEF</td>
<td>$BCC$</td>
<td>04</td>
</tr>
<tr>
<td>9</td>
<td>0000160</td>
<td>OPCDEF</td>
<td>$BCS$</td>
<td>04</td>
</tr>
<tr>
<td>10</td>
<td>0000170</td>
<td>OPCDEF</td>
<td>$BEQ$</td>
<td>04</td>
</tr>
<tr>
<td>11</td>
<td>0000200</td>
<td>OPCDEF</td>
<td>$BGE$</td>
<td>04</td>
</tr>
<tr>
<td>12</td>
<td>0000210</td>
<td>OPCDEF</td>
<td>$BGT$</td>
<td>04</td>
</tr>
<tr>
<td>13</td>
<td>0000220</td>
<td>OPCDEF</td>
<td>$BHI$</td>
<td>04</td>
</tr>
<tr>
<td>14</td>
<td>0000230</td>
<td>OPCDEF</td>
<td>$BHIS$</td>
<td>04</td>
</tr>
<tr>
<td>15</td>
<td>0000240</td>
<td>OPCDEF</td>
<td>$BIC$</td>
<td>02</td>
</tr>
<tr>
<td>16</td>
<td>0000250</td>
<td>OPCDEF</td>
<td>$BICB$</td>
<td>02</td>
</tr>
<tr>
<td>17</td>
<td>0000260</td>
<td>OPCDEF</td>
<td>$BIS$</td>
<td>02</td>
</tr>
<tr>
<td>18</td>
<td>0000270</td>
<td>OPCDEF</td>
<td>$BISB$</td>
<td>02</td>
</tr>
<tr>
<td>19</td>
<td>0000300</td>
<td>OPCDEF</td>
<td>$BIT$</td>
<td>02</td>
</tr>
<tr>
<td>20</td>
<td>0000310</td>
<td>OPCDEF</td>
<td>$BITB$</td>
<td>02</td>
</tr>
<tr>
<td>21</td>
<td>0000320</td>
<td>OPCDEF</td>
<td>$BLE$</td>
<td>04</td>
</tr>
<tr>
<td>22</td>
<td>0000330</td>
<td>OPCDEF</td>
<td>$BLO$</td>
<td>04</td>
</tr>
<tr>
<td>23</td>
<td>0000340</td>
<td>OPCDEF</td>
<td>$BLOS$</td>
<td>04</td>
</tr>
<tr>
<td>24</td>
<td>0000350</td>
<td>OPCDEF</td>
<td>$BLT$</td>
<td>04</td>
</tr>
<tr>
<td>25</td>
<td>0000360</td>
<td>OPCDEF</td>
<td>$BMI$</td>
<td>04</td>
</tr>
<tr>
<td>26</td>
<td>0000370</td>
<td>OPCDEF</td>
<td>$BNE$</td>
<td>04</td>
</tr>
<tr>
<td>27</td>
<td>0000400</td>
<td>OPCDEF</td>
<td>$BPL$</td>
<td>04</td>
</tr>
<tr>
<td>28</td>
<td>0000420</td>
<td>OPCDEF</td>
<td>$BR$</td>
<td>04</td>
</tr>
<tr>
<td>29</td>
<td>0000430</td>
<td>OPCDEF</td>
<td>$BVC$</td>
<td>04</td>
</tr>
<tr>
<td>30</td>
<td>0000440</td>
<td>OPCDEF</td>
<td>$BVS$</td>
<td>04</td>
</tr>
<tr>
<td>31</td>
<td>0000450</td>
<td>OPCDEF</td>
<td>$CCC$</td>
<td>00</td>
</tr>
<tr>
<td>32</td>
<td>0000470</td>
<td>OPCDEF</td>
<td>$CLC$</td>
<td>00</td>
</tr>
<tr>
<td>34</td>
<td>0000500</td>
<td>OPCDEF</td>
<td>$CLN$</td>
<td>00</td>
</tr>
<tr>
<td>35</td>
<td>0000510</td>
<td>OPCDEF</td>
<td>$CLR$</td>
<td>01</td>
</tr>
<tr>
<td>36</td>
<td>0000520</td>
<td>OPCDEF</td>
<td>$CLRB$</td>
<td>01</td>
</tr>
<tr>
<td>39</td>
<td>0000550</td>
<td>OPCDEF</td>
<td>$CLV$</td>
<td>00</td>
</tr>
<tr>
<td>40</td>
<td>0000560</td>
<td>OPCDEF</td>
<td>$CLZ$</td>
<td>00</td>
</tr>
<tr>
<td>Address</td>
<td>Opcode</td>
<td>Instruction</td>
<td>Address</td>
<td>Opcode</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>-------------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>000570</td>
<td>OPCDEF</td>
<td>&lt;CMP&gt;</td>
<td>02</td>
<td>020000</td>
</tr>
<tr>
<td>000600</td>
<td>OPCDEF</td>
<td>&lt;CMPB&gt;</td>
<td>02</td>
<td>120000</td>
</tr>
<tr>
<td>000630</td>
<td>OPCDEF</td>
<td>&lt;COM&gt;</td>
<td>01</td>
<td>005100</td>
</tr>
<tr>
<td>000640</td>
<td>OPCDEF</td>
<td>&lt;COMB&gt;</td>
<td>01</td>
<td>105100</td>
</tr>
<tr>
<td>000650</td>
<td>OPCDEF</td>
<td>&lt;DEC&gt;</td>
<td>01</td>
<td>005300</td>
</tr>
<tr>
<td>000660</td>
<td>OPCDEF</td>
<td>&lt;DECB&gt;</td>
<td>01</td>
<td>105300</td>
</tr>
<tr>
<td>000670</td>
<td>OPCDEF</td>
<td>&lt;EMT&gt;</td>
<td>06</td>
<td>104000</td>
</tr>
<tr>
<td>000730</td>
<td>OPCDEF</td>
<td>&lt;HALT&gt;</td>
<td>00</td>
<td>000000</td>
</tr>
<tr>
<td>000740</td>
<td>OPCDEF</td>
<td>&lt;INC&gt;</td>
<td>01</td>
<td>005200</td>
</tr>
<tr>
<td>000750</td>
<td>OPCDEF</td>
<td>&lt;INCB&gt;</td>
<td>01</td>
<td>105200</td>
</tr>
<tr>
<td>000760</td>
<td>OPCDEF</td>
<td>&lt;TOT&gt;</td>
<td>00</td>
<td>000004</td>
</tr>
<tr>
<td>000770</td>
<td>OPCDEF</td>
<td>&lt;JMP&gt;</td>
<td>01</td>
<td>000100</td>
</tr>
<tr>
<td>001000</td>
<td>OPCDEF</td>
<td>&lt;JSR&gt;</td>
<td>05</td>
<td>004000</td>
</tr>
<tr>
<td>001010</td>
<td>OPCDEF</td>
<td>&lt;MOV&gt;</td>
<td>02</td>
<td>010000</td>
</tr>
<tr>
<td>001230</td>
<td>OPCDEF</td>
<td>&lt;MOVB&gt;</td>
<td>02</td>
<td>110000</td>
</tr>
<tr>
<td>001240</td>
<td>OPCDEF</td>
<td>&lt;NEG&gt;</td>
<td>01</td>
<td>005400</td>
</tr>
<tr>
<td>001320</td>
<td>OPCDEF</td>
<td>&lt;NEGB&gt;</td>
<td>01</td>
<td>105400</td>
</tr>
<tr>
<td>001330</td>
<td>OPCDEF</td>
<td>&lt;NOP&gt;</td>
<td>00</td>
<td>000240</td>
</tr>
<tr>
<td>001360</td>
<td>OPCDEF</td>
<td>&lt;RESET&gt;</td>
<td>00</td>
<td>000005</td>
</tr>
<tr>
<td></td>
<td>Address</td>
<td>Opcode</td>
<td>Instruction</td>
<td>Address</td>
</tr>
<tr>
<td>----</td>
<td>---------</td>
<td>------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>1</td>
<td>001400</td>
<td>OPCDEF</td>
<td>&lt;ROL &gt;</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>001410</td>
<td>OPCDEF</td>
<td>&lt;ROLB &gt;</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>001420</td>
<td>OPCDEF</td>
<td>&lt;ROR &gt;</td>
<td>01</td>
</tr>
<tr>
<td>4</td>
<td>001430</td>
<td>OPCDEF</td>
<td>&lt;RORB &gt;</td>
<td>01</td>
</tr>
<tr>
<td>5</td>
<td>001440</td>
<td>OPCDEF</td>
<td>&lt;RTI &gt;</td>
<td>00</td>
</tr>
<tr>
<td>6</td>
<td>001450</td>
<td>OPCDEF</td>
<td>&lt;RTS &gt;</td>
<td>03</td>
</tr>
<tr>
<td>7</td>
<td>001470</td>
<td>OPCDEF</td>
<td>&lt;SBC &gt;</td>
<td>01</td>
</tr>
<tr>
<td>8</td>
<td>001500</td>
<td>OPCDEF</td>
<td>&lt;SBCB &gt;</td>
<td>01</td>
</tr>
<tr>
<td>9</td>
<td>001510</td>
<td>OPCDEF</td>
<td>&lt;SEC &gt;</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>001520</td>
<td>OPCDEF</td>
<td>&lt;SEN &gt;</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>001530</td>
<td>OPCDEF</td>
<td>&lt;SEV &gt;</td>
<td>00</td>
</tr>
<tr>
<td>12</td>
<td>001600</td>
<td>OPCDEF</td>
<td>&lt;SEZ &gt;</td>
<td>00</td>
</tr>
<tr>
<td>13</td>
<td>001610</td>
<td>OPCDEF</td>
<td>&lt;SUB &gt;</td>
<td>02</td>
</tr>
<tr>
<td>14</td>
<td>002020</td>
<td>OPCDEF</td>
<td>&lt;SWAB &gt;</td>
<td>01</td>
</tr>
<tr>
<td>15</td>
<td>002050</td>
<td>OPCDEF</td>
<td>&lt;TRAP &gt;</td>
<td>06</td>
</tr>
<tr>
<td>16</td>
<td>002070</td>
<td>OPCDEF</td>
<td>&lt;TST &gt;</td>
<td>01</td>
</tr>
<tr>
<td>17</td>
<td>002100</td>
<td>OPCDEF</td>
<td>&lt;TSTB &gt;</td>
<td>01</td>
</tr>
<tr>
<td>18</td>
<td>002110</td>
<td>OPCDEF</td>
<td>&lt;WAIT &gt;</td>
<td>00</td>
</tr>
</tbody>
</table>

C-4
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>002160</td>
<td>DIRDEF</td>
<td>&lt;ASCII&gt;</td>
<td>DFLGBM</td>
</tr>
<tr>
<td>2</td>
<td>002170</td>
<td>DIRDEF</td>
<td>&lt;ASCIIZ&gt;</td>
<td>DFLGBM</td>
</tr>
<tr>
<td>3</td>
<td>002210</td>
<td>DIRDEF</td>
<td>&lt;BLKB&gt;</td>
<td>DFLGEV</td>
</tr>
<tr>
<td>4</td>
<td>002220</td>
<td>DIRDEF</td>
<td>&lt;BLWK&gt;</td>
<td>DFLGBM</td>
</tr>
<tr>
<td>5</td>
<td>002230</td>
<td>DIRDEF</td>
<td>&lt;BYTE&gt;</td>
<td>DFLGBM</td>
</tr>
<tr>
<td>6</td>
<td>002250</td>
<td>DIRDEF</td>
<td>&lt;DSABL&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>7</td>
<td>002260</td>
<td>DIRDEF</td>
<td>&lt;ENABL&gt;</td>
<td>DFLMAC, XMACRO</td>
</tr>
<tr>
<td>8</td>
<td>002270</td>
<td>DIRDEF</td>
<td>&lt;END&gt;</td>
<td>DFLMAC, XMACRO</td>
</tr>
<tr>
<td>9</td>
<td>002300</td>
<td>DIRDEF</td>
<td>&lt;ENDC&gt;</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>002310</td>
<td>DIRDEF</td>
<td>&lt;ENDM&gt;</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>002320</td>
<td>DIRDEF</td>
<td>&lt;ENDR&gt;</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>002340</td>
<td>DIRDEF</td>
<td>&lt;ERROR&gt;</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>002350</td>
<td>DIRDEF</td>
<td>&lt;EVEN&gt;</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>002420</td>
<td>DIRDEF</td>
<td>&lt;IF&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>15</td>
<td>002430</td>
<td>DIRDEF</td>
<td>&lt;IFDF&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>16</td>
<td>002440</td>
<td>DIRDEF</td>
<td>&lt;IFEQ&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>17</td>
<td>002450</td>
<td>DIRDEF</td>
<td>&lt;IFF&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>18</td>
<td>002460</td>
<td>DIRDEF</td>
<td>&lt;IFG&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>19</td>
<td>002470</td>
<td>DIRDEF</td>
<td>&lt;IFGE&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>20</td>
<td>002500</td>
<td>DIRDEF</td>
<td>&lt;IFGT&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>21</td>
<td>002510</td>
<td>DIRDEF</td>
<td>&lt;IFL&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>22</td>
<td>002520</td>
<td>DIRDEF</td>
<td>&lt;IFLE&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>23</td>
<td>002530</td>
<td>DIRDEF</td>
<td>&lt;IFLT&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>24</td>
<td>002540</td>
<td>DIRDEF</td>
<td>&lt;IFNDF&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>25</td>
<td>002550</td>
<td>DIRDEF</td>
<td>&lt;IFNE&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>26</td>
<td>002560</td>
<td>DIRDEF</td>
<td>&lt;IFNZ&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>27</td>
<td>002570</td>
<td>DIRDEF</td>
<td>&lt;IFT&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>28</td>
<td>002600</td>
<td>DIRDEF</td>
<td>&lt;IFTF&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>29</td>
<td>002610</td>
<td>DIRDEF</td>
<td>&lt;IFZ&gt;</td>
<td>DFLCND</td>
</tr>
<tr>
<td>30</td>
<td>002620</td>
<td>DIRDEF</td>
<td>&lt;IIF&gt;</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>002630</td>
<td>DIRDEF</td>
<td>&lt;IRP&gt;</td>
<td>DFLMAC, XMACRO</td>
</tr>
<tr>
<td>32</td>
<td>002640</td>
<td>DIRDEF</td>
<td>&lt;IRPC&gt;</td>
<td>DFLMAC, XMACRO</td>
</tr>
<tr>
<td>33</td>
<td>002660</td>
<td>DIRDEF</td>
<td>&lt;LIST&gt;</td>
<td></td>
</tr>
<tr>
<td>Line</td>
<td>Address</td>
<td>Symbol</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>--------</td>
<td>------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>1</td>
<td>002670</td>
<td>DIRDEF</td>
<td><code>&lt;MACR&gt;</code></td>
<td>DFLMAC, XMACRO</td>
</tr>
<tr>
<td>2</td>
<td>002700</td>
<td>DIRDEF</td>
<td><code>&lt;MACRO&gt;</code></td>
<td>DFLMAC, XMACRO</td>
</tr>
<tr>
<td>3</td>
<td>002720</td>
<td>DIRDEF</td>
<td><code>&lt;MEXIT&gt;</code></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>002730</td>
<td>DIRDEF</td>
<td><code>&lt;NARG&gt;</code></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>002740</td>
<td>DIRDEF</td>
<td><code>&lt;NCHR&gt;</code></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>002750</td>
<td>DIRDEF</td>
<td><code>&lt;NLIST&gt;</code></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>002760</td>
<td>DIRDEF</td>
<td><code>&lt;NTYPE&gt;</code></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>002770</td>
<td>DIRDEF</td>
<td><code>&lt;OCC&gt;</code></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>003000</td>
<td>DIRDEF</td>
<td><code>&lt;PAGE&gt;</code></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>003010</td>
<td>DIRDEF</td>
<td><code>&lt;PRINT&gt;</code></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>003020</td>
<td>DIRDEF</td>
<td><code>&lt;RADIX&gt;</code></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>003030</td>
<td>DIRDEF</td>
<td><code>&lt;RAD50&gt;</code></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>003040</td>
<td>DIRDEF</td>
<td><code>&lt;REM&gt;</code></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>003050</td>
<td>DIRDEF</td>
<td><code>&lt;REPT&gt;</code></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>003060</td>
<td>DIRDEF</td>
<td><code>&lt;SBTTL&gt;</code></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>003070</td>
<td>DIRDEF</td>
<td><code>&lt;TITLE&gt;</code></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>003100</td>
<td>WRDSYM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>003100</td>
<td>DIRDEF</td>
<td><code>&lt;WORD&gt;</code></td>
<td>DFLGEV</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>003110</td>
<td>PSTTOP</td>
<td></td>
<td><code>TOP LIMIT</code></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>000001</td>
<td>.END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Addressing error. An address within the instruction is incorrect. Also may indicate a relocation error. This message does not necessarily reflect a coding error.</td>
</tr>
<tr>
<td>B</td>
<td>Bounding error. Instructions or word data are being assembled at an odd address in memory. The location counter is updated by +1.</td>
</tr>
<tr>
<td>D</td>
<td>Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.</td>
</tr>
<tr>
<td>E</td>
<td>End directive not found. (A listing is generated.)</td>
</tr>
<tr>
<td>I</td>
<td>Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing. The character is then ignored.</td>
</tr>
<tr>
<td>L</td>
<td>Line buffer overflow; i.e., input line greater than 132 characters. Extra characters on a line (more than 72(10)) are ignored.</td>
</tr>
<tr>
<td>M</td>
<td>Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a previously encountered label.</td>
</tr>
<tr>
<td>N</td>
<td>Number containing 8 or 9 has decimal point missing.</td>
</tr>
<tr>
<td>P</td>
<td>Phase error. A label's definition of value varies from one pass to another.</td>
</tr>
<tr>
<td>Q</td>
<td>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.</td>
</tr>
</tbody>
</table>
Register-type error. An invalid use of or reference to a register has been made.

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.

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.

Instruction which is not compatible among all members of the PDP-11 family (11/15, 11/20, and 11/45).
INDEX

Absolute mode, 4-5
Addressing modes, 4-1
  branch instruction, 4-7
  preferred, 1-6
  syntax, B-2
.ASCII directive, 5-11
.ASCIIZ directive, 5-12
Assembler directives, 5-1, B-10
Assembly language and assembler, B-1
Assembly instructions, B-3
  branch, B-8
  double-operand, B-4
  operator, B-6
  single-operand, B-4
  subroutine return, B-9
  trap, B-7
Assembly location counter, 3-10
Assembly listing, example, 7-3
Autodecrement deferred mode, 4-3
Autodecrement mode, 4-3
Autoincrement deferred mode, 4-3
Autoincrement mode, 4-2
Automatically created symbols, 6-7

.BLKB and .BLKW directives, 5-17
Branch instruction addressing, 4-7
Branch instructions, conditional, 1-8
.BYTE directive, 5-8

Character set, 3-1
  illegal characters, 3-4
  MACII, A-1
  operator characters, 3-4
  RADIX-50, A-4
Command input string, 7-1
Comments within programs, 1-4
  field, 2-3

Concatenation, 6-8
Conditional assemblies, 1-6
  directives, 5-18
Conversion (ASCII) of one or
two characters, 5-10

Delimiters, 3-2
Direct assignment statements, 3-6
  .DSABL directive, 5-7

.ERROR and .PRINT directives, 6-10
  .END directive, 5-17
  .ENDM directive, 6-2
  Error codes, D-1
  .EVEN directive, 5-15
  Expressions, 3-12

Format control, 2-4
Immediate conditional directives, 5-21
  Immediate mode, 4-4
  Indefinite repeat block (.IRP and
  .IRPC) directives, 6-11
  Index deferred mode, 4-4
  Index mode, 4-4

Label field, 2-2
Listing control directives, 5-1
Listing, MACII example, 5-4
Loading MACII, 7-1
Location counter, 5-15
MACRO calls, 6-3
MACRO definition, 6-1
    arguments, 6-4
    formatting, 6-3
MACRO directives, 6-1
MACRO nesting, 6-4
.MACRO directive, 6-1
.MEXIT directive, 6-2
Mode forms and codes, table, 4-6
Modular programming, 1-1

Registers
    increment, 1-8
    localized usage, 1-4
    register deferred mode, 4-2
    register mode, 4-2
    register symbols, 3-7
Relative deferred mode, 4-5
Relative mode, 4-5
Repeat block (.REPT) directive, 6-14

.SBTTL directive, 5-5
Source program format, 2-1
Special characters, 6-5, B-1
Statement format, 2-1
direct assignment, 3-6
Subconditional directives, 5-19
Symbols
    MAC1l, 3-5
    local, 3-8
    permanent, 3-5
    register, 3-7
    user-defined & MACRO, 3-5

Page ejection, 5-7
Page headings, 5-5
PAL-11R conditional assembly
directives, 5-22
Parameter assignments, 1-7
Permanent symbol table, C-1
.PRINT and .ERROR directives, 6-10
.WORD directive, 5-9

_RAD50 directive, 5-12
_RADIX control directive, 5-14
temporary, 5-14
Reentrant code, 1-6

.I-2
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
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. ☐