MACRO-11 ASSEMBLER PROGRAMMER'S MANUAL

MACRO-11 ASSEMBLER
PROGRAMMER'S MANUAL

Macro Assembly Language
and
Relocatable Assembler
for the
Disk Operating System

June 1972

SOFTWARE SUPPORT CATEGORY

The software described in this document is supported by DEC unter category I, defined on page iii of this document.

For additional copies, order No. DEC-11-OMACA-A-D from Digital Equipment Corporation, Software Distribution Center, Building 1-2, Maynard, Massachusetts, 01754

DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS

First Printing, April 1972
Second Printing, June 1972

Your attention is invited to the last two pages of this document. The "How to Obtain Software Information" page tells you how to keep up-to-date with DEC's software. The "Reader's Comments" page when filled in and mailed, is beneficial to both you and DEC; any comments received are acknowledged and are considered when documenting subsequent manuals.

Associated Documents:

PDP-11/20 Processor Handbook 112.01071.1855

PDP-11/45 Processor Handbook 112.01071.1876

PDP-11 Peripherals and Interfacing Handbook
112.01071.1854

PDP-11 Disk Operating System Monitor
Programmer's Handbook, DEC-11-MWDB-D

PDP-11 Batch User's Guide, DEC-11-OBUDA-A-D

PDP-11 Edit-11 Text Editor
Programmer's Manual, DEC-11-EEDA-D

PDP-11 ODT-11R Debugging Program
Programmer's Manual, DEC-11-OODA-D

PDP-11 PIP, File Utility Package,
Programmer's Manual, DEC-11-PIDB-D

PDP-11 Link-11 Linker and Libr-11 Librarian
Programmer's Manual, DEC-11-ZLDB-D

PREFACE

This manual describes the PDP-11 MACRO-11 Assembler and Assembly Language. It also describes, in brief, 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 document (although this document does stand complete by itself, 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 DEC Software Distribution Center.)

MACRO-11 operates under the PDP-11 DOS (Disk Operating System) Monitor and the PDP-11 BATCH Monitor.

Some notable features of MACRO-11 are:

1. Program and command string control of assembly functions.
2. Device and file name specifications for input and output files
3. Error listing on command output device
4. Double buffered and concurrent I/O
5. Alphabetized, formatted symbol table listing
6. Relocatable object modules
7. Global symbols for linking between object modules
8. Conditional assembly directives
9. Program sectioning directives
10. User defined macros
11. Comprehensive set of system macros
12. Extensive listing control
13. Symbolic cross referencing.

SOFTWARE SUPPORT CATEGORIES

Digital Equipment Corporation (DEC) makes available four categories of software. These categories reflect the types of support a customer may expect from DEC for a specified software product. DEC reserves the right to change the category of a software product at any time. The four categories are as follows:

CATEGORY I
Software Products Supported at no Charge

This classification includes current versions of monitors, programming languages, and support programs provided by DEC. DEC will provide installation (when applicable), advisory, and remedial support at no charge. These services are limited to original purchasers of DEC computer systems who have the requisite DEC equipment and software products.

At the option of DEC, a software product may be recategorized from Category I to Category II for a particular customer if the software product has been modified by the customer or a third party.

CATEGORY II
Software Products that Receive Support for a Fee

This category includes prior versions of Category I programs and all other programs available from DEC for which support is given. Programming assistance (additional support), as available, will be provided on these DEC programs and non-DEC programs when used in conjunction with these DEC programs and equipment supplied by DEC.

CATEGORY III
Pre-Release Software

DEC may elect to release certain software products to customers in order to facilitate final testing and/or customer familiarization. In this event, DEC will limit the use of such pre-release software to internal, non-competitive applications. Category III software is only supported by DEC where this support is consistent with evaluation of the software product. While DEC will be grateful for the reporting of any criticism and suggestions pertaining to a pre-release, there exists no commitment to respond to these reports.

CATEGORY IV
Non-Supported Software

This category includes all programs for which no support is given.

CONTENTS

PART I

INTRODUCTION TO MACRO-11

PART II

DETAILS ON PROGRAMMING IN MACRO-11

PART IV

OPERATING PROCEDURES

CHAPTER 1

FUNDAMENTALS OF PROGRAMMING THE PDP-11

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

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

No attempt is made in this document to describe the PDP-11 hardware or the function of the various PDP-11 instructions. The reader is advised to 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 these are the techniques found to work best with the PDP-11 and are used in PDP-11 system programs, it is advisable to be considering them from the very start of your PDP-11 programming experience.

1.1 MODULAR PROGRAMMING

The PDP-11 family of computers lend themselves most easily to a modular system of programming. In such a system the programmer must envision the entire program and break it down into constituent subroutines. This will provide for the best use of the PDP-11 hardware (as will become clearer later in this Chapter), as well as resulting 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 attack small bits of the

program at any one 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 where similar or identical functions are required without the overhead of redundant development. Software modules devloped as functional entities are more likely to be free of serious logical errors as a result of the original programming effort. Confidence in such modules allows for easy creation of later systems incorporating proven pieces.

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

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.

It is assumed that the human mind can best work with limited pieces of information at any one time, combining the results of the individual functions to encompass the entire program in steps. 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 functions. (The Assembler itself is a tree structure and is briefly described in Figure 1-1.)

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

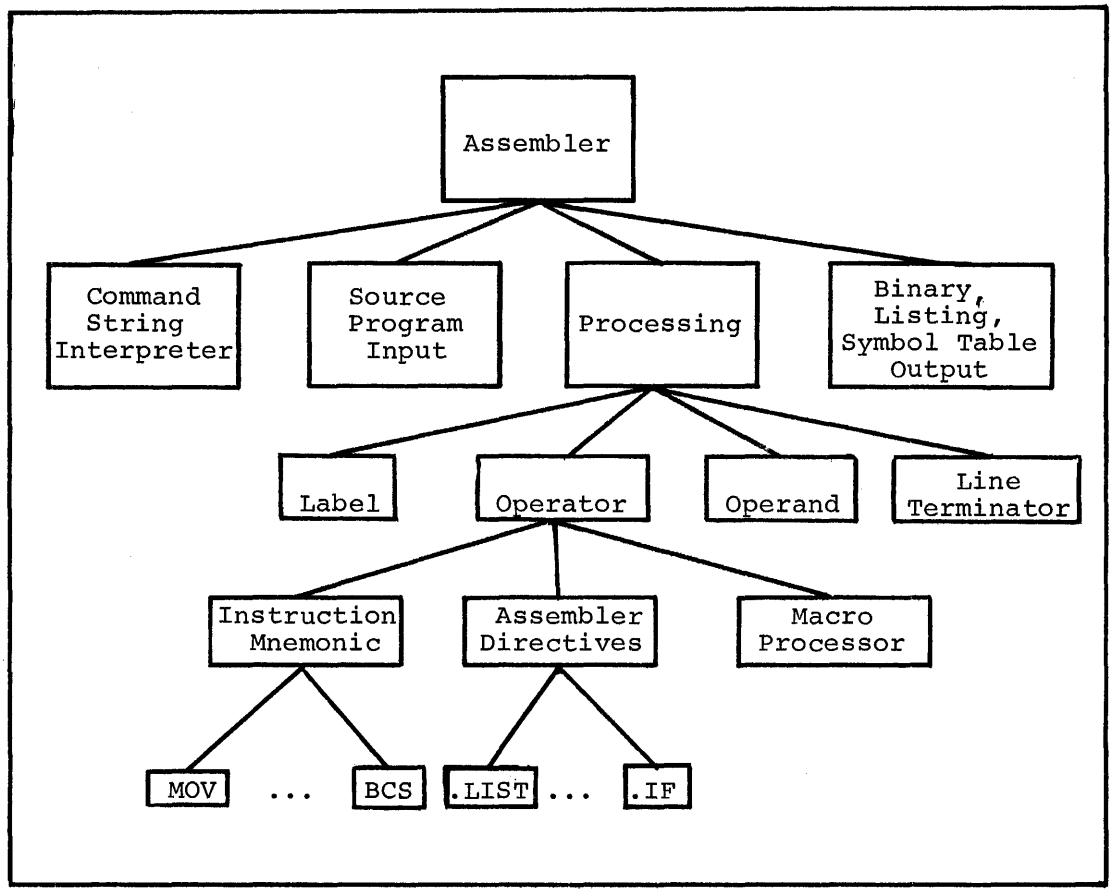


Figure 1-1 Problem Oriented Tree-Structure

what it does. The how should be documented only when the procedure is not obvious to the reader. This enables any later inspection of the 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 two and three-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.

1.1.3 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 6.11.)

Conditional assemblies can also be used to generate 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:

a. 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.

```
1                                       .TFT
2  002060                10$:   CALL    20$                 ;MOVE A CHARACTER
3  002064 003375                BGT     10$                 ;LOOP IF GT ZERO
4  002066 001432                BEQ     19$                 ;END IF ZERO
5  002070 114200                MOVB    *(R2),R0            ;TERMINATOR, BACK UP POINTER
6  002072 020027                CMP     R0,#MT.MAX          ;END OF TYPE?
           177603
7  002076 101453                BLOS    22$                 ;  YES
8  002100 010146                MOV     R1,-(SP)            ;REMEMBER READ POINTER
9  002102 016701                MOV     MSBARG,R1
           002034'
10 02106 005721                 TST     (R1)+
11 02110 010203                 MOV     R2,R3               ;  AND WRITE POINTER
12 02112 005400                 NEG     R0                  ;ASSUME MACRO
13 02114 026727                 CMP     MSBTYP,#MT.MAC      ;TRUE?
           002026'
           177603
14 02122 001402                 BEQ     12$                 ;  YES, USE IT
15 02124 016700                 MOV     MSBCNT,R0           ;GET ARG NUMBER
           002036'
16 02130 010302 12$:            MOV     R3,R2               ;RESET WRITE POINTER
17 02132                13$:    CALL    20$                 ;MOVE A BYTE
18 02136 003375                 BGT     13$                 ;LOOP IF PNZ
19 02140 002402                 BLT     14$                 ;END IF LESS THAN ZERO
20 02142 005300                 DEC     R0                  ;ARE WE THERE YET?
21 02144 003371                 BGT     12$                 ;  NO
22 02146 105742 14$:            TSTB    *(R2)               ;YES, BACK UP POINTER
23 02150 012601                 MOV     (SP)+,R1            ;RESET READ POINTER
24 02152 000742                 BR      10$                 ;END OF ARGUMENT SUBSTITUTION
25
26 02154 010167 19$:            MOV     R1,MSBMRP           ;END OF LINE, SAVE POINTER
           002042'
27 02160 052767                 BIS     #LC.ME,LCFLAG       ;FLAG AS MACRO EXPANSION
           000400
           000010'
28 02166 000726                 BR      9$
29
30 02170 032701 20$:            BIT     #BPMB-1,R1          ;MACRO, END OF BLOCK?
           000017
31 02174 001003                 BNE     21$                 ;  NO
32 02176 016101                 MOV     -BPMB(R1),R1        ;YES, POINT TO NEXT BLOCK
           177760
33 02202 005721                 TST     (R1)+               ;MOVE PAST LINK
34 02204 020227 21$:            CMP     R2,#LINBUF+SRCLEN       ;OVERFLOW?
           001744'
35 02210 101404                 BLOS    23$                 ;  NO
36 02212                        ERROR   L                   ;YES, FLAG ERROR
37 02220 105742                 TSTB    *(R2)               ;  AND MOVE POINTER BACK
38 02222 112122 23$:            MOVB    (R1)+,(R2)+         ;MOVE CHAR INTO LINE BUFFER
39 02224                        RETURN
40
41 02226                22$:    CALL    ENDMAC              ;CLOSE MACRO
42 02232 000167                 JMP     1$
           177326
43                              .ENDC
44
45
```

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

b. 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.

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

1.2 POSITION INDEPENDENT CODE (PIC)

NOTE

As this Section is quite detailed, it may be bypassed in the initial reading of the manual.

The output of MACRO-11 (and PAL-11R) assemblies is a relocatable object module. This module, under DOS, is linked (with Link-11) to a specified address prior to being executed.

Once linked, a program can generally be loaded and executed only at the address specified at link time. This is because the Linker has had to make adjustments in some lines to reflect the absolute area of core (locations) in which the program is to run.

It is possible to write a source program than can be loaded and run in any section of core. Such a program is said to consist of position independent code. The construction of position independent code is dependent upon the correct usage of PDP-11 addressing modes. (Addressing modes are described in detail in Chapter 5. The remainder of this Section assumes the reader is familiar with the various addressing modes.)

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

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

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

The relative addressing modes are generally position independent. These modes are as follows:

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

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

Index modes can be either position independent or nonposition independent, according to their usage in the program. These modes are:

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

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

```
MOV 2(SP),RØ      ;POSITION INDEPENDENT
N=4
MOV N(SP),RØ      ;POSITION INDEPENDENT
CLR ADDR(R1)      ;NONPOSITION  INDEPENDENT
```

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

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

```
#N          immediate mode
```

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

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

```
@#A         absolute mode
```

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

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

Position independent code is used in writing programs such as device drivers and utility routines which are most useful when they can be brought into any available core space. Figure 1-3 and Figure 1-4 show pieces of device driver code; one of which is position independent and the other is not.

```
;DVRINT -- ADDRESS OF DEVICE DRIVER INTERRUPT SERVICE
;VECTOR -- ABSOLUTE ADDRESS OF DEVICE INTERRUPT VECTOR
;DRIVER -- START ADDRESS OF DEVICE DRIVER
MOV    #DVRINT, VECTOR     ;SET INTERRUPT ADDRESS
MOVB   DRIVER+6,VECTOR+2   ;SET PRIORITY
CLRB   VECTOR+3            ;CLEAR UPPER STATUS BYTE
```

Figure 1-3 Non-Position Independent Code

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

Figure 1-4 Position Independent Code

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

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

The instruction to initialize the driver priority level uses

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

In the position independent code, operations are performed in registers wherever possible. The process of initializing registers is carefully planned to be position independent. For example: the first two instructions obtain the starting address of the driver. The current PC value is loaded into R1, and the offset from the start of the driver to the current location is added to that value. Each of these operations is position independent. The immediate mode value of VECTOR is loaded into R2; which places the absolute address of the transfer vector into a register for later use. The transfer vector is then cleared, and the offset from the driver starting address is loaded into the vector. The starting address of the driver is then added into the vector, giving the desired entry point to the driver. (This is equivalent to the first statement in Figure 1-3.) Since R2 has been updated to point to VECTOR+2, that location is then cleared and the priority level inserted into the appropriate byte.

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

The MACRO-11 Assembler provides the user with a way of checking the position independence of his code. In an assembly listing, MACRO-11 inserts a ' character following the contents of any word which requires the Linker to perform an operation. In some cases this character indicates a nonposition independent instruction, in other cases, it merely draws the user's attention to the use of a symbol which may or may not be position independent. The cases which cause a ' character in the assembly listing are as follows:

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

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

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

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

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

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

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

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

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

Examples of assembly listings contining the ' character are shown below:

```
1  011744          ENDP2:                                  ;END OF PASS 2
2                          .IF NDF XCREF
3  011744 016702           MOV     CREFNT,R2               ;ANY CREF IN PROGRESS?
          000142'
4  011750 001402           BEQ     8$                      ;  NO
5  011752                  CALL    CREDMP                  ;YES, DUMP AND CLOSE BUFFER
6  011756          8$:
7                          .ENDC
8  011756 005767           TST     BLKTYP                  ;ANY OBJECT OUTPUT?
          000542'
9  011762 001423           BEQ     1$                      ;  NO
10 11764                   CALL    OBJDMP                  ;YES, DUMP IT
11 11770 012767            MOV     #BLKT06,BLKTYP          ;SET END
          000006
          000542'
12 11776                   CALL    RLDDMP                  ;DUMP IT
13                         .IF NDF XEDABS
14 12002 032767            BIT     #ED.ABS,EDMASK          ;ABS OUTPUT?
          000002
          000124'
15 12010 001010            BNE     1$                      ;  NO
16 12012 016700            MOV     OBJPNT,R0
          000536'
17 12016 016720            MOV     ENDVEC+6,(R0)+          ;SET END VECTOR
          000044'
18 12022 010067            MOV     R0,OBJPNT
          000536'
19 12026                   CALL    OBJDMP
20                         .ENDC
21 12032 105767 1$:        TSTB    LSTPL+2                 ;ANY LISTING OUTPUT?
          000546'
22 12036 001474            BEQ     15$                     ;  NO
23 12040 032767            BIT     #LC.SYM,LCMASK          ;SYMBOL TABLE SUPPRESSION?
          040000
          000110'
```

```
24 12046 001070          BNE     15$             ;  YES
25 12050 005067          CLR     LPPCNT          ;FORCE NEW PAGE
         000010'
26 12054 005067          CLR     RCLLPD          ;SET FOR SYMBOL TABLE SCAN
         000006'
27 12060 012702 2$:      MOV     #LINBUF,R2      ;POINT TO STORAGE
         001540'
28 12064         3$:     NEXT    SYMBOL          ;GET THE NEXT SYMBOL
29 12074 001456          BEQ     20$             ;  NO MORE
30 12076                 RSUNP                   ;UNPACK THE SYMBOL
31 12102 012703          MOV     #ENDP2T,R3
         012334'
32 12106                 CALL    ENDP2R
33 12112 012701          MOV     #MODE,R1        ;POINT TO MODE BITS
         000006'
34 12116 032711          BIT     #DEFFLG,(R1)    ;DEFINED?
         000010
35 12122 001403          BEQ     4$              ;  NO
36 12124                 CALL    SFTWRD
37 12130 000404          BR      6$
38
39 12132 012701 4$:      MOV     #STARS,R1
```

1.3 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.

1.4 PREFERRED ADDRESSING MODES

Addressing modes are described in detail in Chapter 5. 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 cost up to three words.

For example:

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

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

1.5 PARAMETER ASSIGNMENTS

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

```
SYM=42
 .
 .
 .
MOV #SYM,RØ
```

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

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

The PDP-11/45 floating-point accumulators are named by convention as follows:

```
ACØ = %Ø
AC1 = %1
AC2 = %2
AC3 = %3
AC4 = %4
AC5 = %5
```

Use of these standard symbols makes examination of another programmer's code much easier than the use of random symbolic names or constants which do not appear on CREF listings.

NOTE

Where a register reference is made in a 2-bit field within a floating-point instruction, ACØ through AC3 may be referenced. In such instructions the 6-bit source or destination field can be filled with addressing modes 1 through 7 which reference the processor registers RØ through R7 or addressing mode Ø which references floating-point registers ACØ through AC5.

1.6 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.

1.6.1 Trap Handler

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

1. A JSR instruction which generally requires two PDP-11 words:
   ```
   JSR R5,SUBA
   ```
   but is direct and fast.
2. A TRAP instruction which requires one PDP-11 word:
   ```
   TRAP N
   ```
   but is indirect and slower. The TRAP handler must use N to index through a dispatch table of subroutine addresses and then JMP to the Nth subroutine in the table.

1.6.2 Register Increment

The operation:

```
CMPB (RØ)+,(RØ)+
```

is preferable to;

```
TST (RØ)+
```

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

1.7 CONDITIONAL BRANCH INSTRUCTIONS

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

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

A common pitfall 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 ($1ØØØØØ_8$) bound. This type of coding error usually appears as a result of re-linking at different addresses and/or a change in 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 Edit-11 automatically appends a line feed to every carriage return character, 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 132_{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 MACRO-11 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 RØ
```

while others have two, for example:

```
MOV #344,R2
```

An assembly language statement must be complete on one source line. No continuation lines are allowed. (If a continuation is attempted with a line feed under Edit-11 the Assembler interprets this as the statement terminator.)

MACRO-11 source statements are formatted with Edit-11 such that use of the TAB character causes the statement fields to be aligned. For example:

```
 Label    Operator        Operand                  Comment
 Field    Field           Field                    Field
MASK=-1Ø
REGEXP:                                    ;REGISTER EXPRESSION
          ABSEXP                           ;MUST BE ABSOLUTE
REGTST:   BIT      #MASK,VALUE             ;3 BITS?
          BEQ      REGERX                  ;YES, OK
REGERR:   ERROR    R                       ;NO, ERROR
REGERX:   MOV      #DEFFLG!REGFLG,MODE
          BIC      #MASK,VALUE
          BR       ABSERX
```

2.1.1 Label Field

A label is a user-defined symbol which is assigned the value of the current location counter and entered into the user-defined symbol table. The value of the label may be either absolute or relocatable, depending on whether the location counter value is currently absolute or relocatable. In the latter case, the absolute value of the symbol is assigned by Link-11, i.e., the stated relocatable value plus the relocation constant.

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. In this example if the location

counter were relocatable, the final value of ABCD would be 100_8+K, where K is the location of the beginning of the relocatable section in which the label ABCD appears.

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

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 6.3.2).

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. (See also Section 6.1.6 for a description of page formatting with respect to macros and Section 6.1.3 for a description of assembly listing output.)

CHAPTER 3

SYMBOLS AND EXPRESSIONS

This Chapter describes the various components of legal MACRO-11 expressions: the Assembler character set, symbol construction, numbers, operators, terms and expressions.

3.1 CHARACTER SET

The following characters are legal in MACRO-11 source programs:

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

b. The digits Ø through 9.

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

d. The following special characters:

Character	Designation	Function
carriage return		formatting character
line feed form feed vertical tab	}	source statement terminators
:	colon	label terminator
=	equal sign	direct assignment indicator
%	percent sign	register term indicator
tab		item or field terminator
space		item or field terminator
#	number sign	immediate expression indicator
@	at sign	deferred addressing indicator
(	left parenthesis	initial register indicator
)	right parenthesis	terminal register indicator
,	comma	operand field separator
;	semi-colon	comment field indicator
<	left angle bracket	initial argument or expression indicator
>	right angle bracket	terminal argument or expression indicator
+	plus sign	arithmetic addition operator or auto increment indicator
-	minus sign	arithmetic subtraction operator or auto decrement indicator
*	asterisk	arithmetic multiplication operator
/	slash	arithmetic division operator
&	ampersand	logical AND operator
!	exclamation	logical inclusive OR operator
"	double quote	double ASCII character indicator
'	single quote	single ASCII character indicator
↑	up arrow	universal unary operator, argument indicator
\	backslash	macro numeric argument indicator

3.1.1 Separating and Delimiting Characters

Reference is made in the remainder of the manual to legal separating characters and legal argument delimiters. These terms are defined below in Tables 3-1 and 3-2.

TABLE 3-1

Legal Separating Characters

Character	Definition	Usage
space	one or more spaces and/or tabs	A space is a legal separator only for argument operands. Spaces within expressions are ignored (see Section 3.8).
,	comma	A comma is a legal separator for both expressions and argument operands.

TABLE 3-2

Legal Delimiting Characters

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

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

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

indicates a macro definition with two arguments, while

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

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

Angle brackets can be nested as follows:

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

which reduces to:

```
A<B>C
```

and which is considered to be one argument in both forms.

3.1.2 Illegal Characters

A character can be illegal in one of two ways:

a. A character which is not recognized as an element of the MACRO-11 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.

b. A legal MACRO-11 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 MACRO-11 are as follows:

Unary Operator	Explanation		Example
+	plus sign	+A	(positive value of A, equivalent to A)
-	minus sign	-A	(negative, 2's complement, value of A)

Unary Operator	Explanation	Example	
↑	up arrow, universal unary operator (this usage is described in greater detail in Sections 6.4.2 and 6.6.2).	↑F3.Ø	(interprets 3.Ø as a one word floating-point number)
		↑C24	(interprets the one's complement value of 24_8)
		↑D127	(interprets 127 as a decimal number)
		↑O34	(interprets 34 as an octal number)
		↑B11ØØØ111	(interprets 11000111 as a binary value)

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

```
-%5
↑C↑O12
```

Legal binary operators under MACRO-11 are as follows:

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

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

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

3.2 MACRO-11 SYMBOLS

There are three types of symbols: permanent, user-defined and macro. MACRO-11 maintains three types of symbol tables: the Permanent Symbol Table (PST), the User Symbol Table (UST) and the Macro Symbol Table (MST). The PST contains all the permanent symbols and is part of the MACRO-11 Assembler load module. The UST and MST are constructed as the source program is assembled; user-defined symbols are added to the table as they are encountered.

3.2.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics (Appendix B.3) and assembler directives (Chapters 6 and 7, 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 7.1). These symbols are added to the Macro Symbol Table as they are encountered during the assembly.

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

The $ and . characters are reserved for system software symbols (e.g., .READ, a system macro) and should not be inserted in user-defined or macro symbols.

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

a. The first character must not be a number.

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

c. 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.

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

The value of a symbol depends upon its use in the program. A symbol in the operator field may be any one of the three symbol types. To determine the value of the symbol, the Assembler searches the three symbol tables in the following order:

a. Macro Symbol Table
b. Permanent Symbol Table
c. User-Defined Symbol Table

A symbol found in the operand field is sought in the

a. User-Defined Symbol Table
b. 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.

User-defined symbols are either internal or external (global). All user-defined symbols are internal unless explicitly defined as being global with the .GLOBL directive (see Section 6.10).

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

Since MACRO-11 provides program sectioning capabilities (Section 6.9), two types of internal symbols must be considered:

a. symbols that belong to the current program section; and
b. symbols that belong to other program sections.

In both cases, the symbol must be defined within the current assembly; the significance of the distinction is critical in evaluating expressions involving type (b) above (see Section 3.9).

3.3 DIRECT ASSIGNMENT

A direct assignment statement associates a symbol with a value. When a direct assignment statement defines a symbol for the first time, that symbol is entered into the user symbol table 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 relocatable or absolute attribute of their defining expression. However, if the defining expression is global, the symbol is not global unless explicitly defined as such in a .GLOBL directive (see Section 6.10).

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:

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

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

c. Only one symbol can be defined by any one direct assignment statement.

d. Only one level of forward referencing is allowed.

Example of two levels of forward referencing (illegal):

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

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

3.4 REGISTER SYMBOLS

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

```
%Ø
%1
 :
 :
%7
```

where the digit indicating the specific register can be replaced by any legal term which can be evaluated during the first assembly pass. Use of such register designations does not result in any indication of register usage within the CREF listing.

It is recommended that the programmer create and use symbolic names for all register references. A register symbol is defined in a direct assignment statement, among the first statements in the program. The defining expression of a register symbol must be absolute. For example:

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

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

Although its use is not noted in CREF listings, the % character can be used with any term or expression to specify a register. (A register expression less than Ø or greater than 7 is flagged with an R error code.) For example:

```
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 a 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, then, are not referenced from other object moudles or even 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$
1Ø4$
```

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

Local symbols 64$ through 127$ can be generated automatically as a feature of the macro processor (see Section 7.3.5 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:

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

   LABEL=.

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

b. The range of a local symbol block is terminated upon encountering a .CSECT directive.

c. The range of a single local symbol block can be delimited with the .ENABL LSB and the first symbolic label or .CSECT directive following the .DSABL LSB directives. The default for LSB is off.

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

```
Line    Octal
Number  Expansion          Source Code                    Comments
  1                        .SBTTL  SECTOR INTTTALIZATION
  2
  3         000000'        .CSECT  IMPURE          ;IMPURE STORAGE AREA
  4 000000            IMPURE:
  5         000000'        .CSECT  IMPPAS          ;CLEARED EACH PASS
  6 000000            IMPPAS:
  7         000000'        .CSECT  IMPLIN          ;CLEARED EACH LINE
  8 000000            IMPLIN:
  9
 10         000000'        .CSECT  XCTPRG          ;PROGRAM INITTIALIZATION CODE
 11 00000            XCTPRG:
 12 00000 012700         MOV     #IMPURE,R0
            000000'
 13 00004 005020 1$:     CLR     (R0)+           ;CLEAR IMPURE AREA
 14 00006 022700         CMP     #IMPTOP,R0
            000040'
 15 00012 101374         BHI     1$
 16
 17         000000'        .CSECT  XCTPAS          ;PASS INITIALIZATION CODE
 18 00000            XCTPAS:
 19 00000 012700         MOV     #IMPPAS,R0
            000000'
 20 00004 005020 1$:     CLR     (R0)+           ;CLEAR IMPURE PART
 21 00006 022700         CMP     #IMPTOP,R0
            000040'
 22 00012 101374         BHI     1$
 23
 24         000000'        .CSECT  XCTLIN          ;LINE INITIALIZATION CODE
 25 00000            XCTLIN:
 26 00000 012700         MOV     #IMPLIN,R0
            000000'
 27 00004 005020 1$:     CLR     (R0)+
 28 00006 022700         CMP     #IMPTOP,R0
            000040'
 29 00012 101374         BHI     1$
 30
 31         000000'        .CSECT  MIXED           ;MIXED MODE SECTOR
```

Figure 3-3

Assembly Source Listing of MACRO-11 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 #.,RØ     ;. REFERS TO LOCATION A,
                  ;I.E., THE ADDRESS OF THE
                  ;MOV INSTRUCTION.
```

(# is explained in Section 5.9).

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

```
.=expression
```

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

The mode of the location counter symbol can be changed by the use of the .ASECT or .CSECT directive as explained in Section 6.9.

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

Examples:

```
          .ASECT
     .=5ØØ                        ;SET LOCATION COUNTER TO ABSOLUTE
                                  ;5ØØ

FIRSt:    MOV  .+1Ø,COUNT        ;THE LABEL FIRST HAS THE VALUE 5ØØ(8)
                                  ;.+1Ø EQUALS 51Ø(8). THE CONTENTS OF
                                  ;THE LOCATION 51Ø(8) WILL BE DEPOSITED
                                  ;IN LOCATION COUNT.

  .=52Ø                           ;THE ASSEMBLY LOCATION COUNTER NOW
                                  ;HAS A VALUE OF ABSOLUTE 52Ø(8).

SECOND:   MOV  .,INDEX            ;THE LABEL SECOND HAS THE VALUE 52Ø(8)
                                  ;THE CONTENTS OF LOCATION 52Ø(8), THAT
                                  ;IS, THE BINARY CODE FOR THE   INSTRUC-
                                  ;TION ITSELF, WILL BE DEPOSITED IN
                                  ;LOCATION INDEX.

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

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

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

```
.=.+1ØØ
```

reserves 100_8 bytes of storage space in the program. The next instruction is stored at 1100.

3.7 NUMBERS

The MACRO-11 Assembler assumes all numbers in the source program are to be interpreted in octal radix unless otherwise specified. The assumed radix can be altered with the .RADIX directive (see Section 6.4.1) or individual numbers can be treated as being of decimal, binary, or octal radix (see Section 6.4.2).

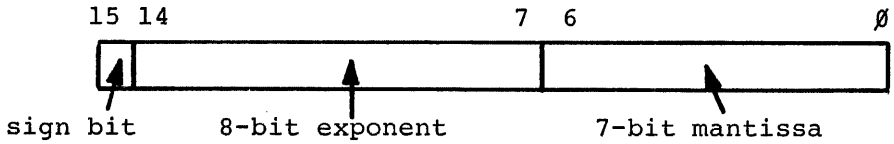
Octal numbers consist of the digits Ø through 7 only. A number not specified as a decimal number and containing an 8 or 9 is flagged with an N error code and treated as a decimal number.

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

A number 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.

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

The single-word floating-point numbers which can be generated with the ↑F operator (see Section 6.4.2) are stored in the following format:



Refer to PDP-11/45 Processor Handbook for details of the floating-point format.

3.8 TERMS

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

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

b. A symbol, as defined earlier in the Chapter. Symbols are interpreted according to the following hierarchy:

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

   2. a permanent symbol whose basic value is used and whose arguments (if any) are ignored,

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

c. 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 6.3.3.)

d. 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 6.3.1) 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). The evaluation of an expression includes the evaluation of the mode of the resultant expression; that is, absolute, relocatable or external. Expression modes are defined further below.

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

```
-+-A
```

is equivalent to:

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

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

```
TAG ! LA 177777
```

is evaluated as

```
TAG ! LA+177777
```

with a Q error flag on the assembly listing line.

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

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

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

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

c. An expression is external (or global) if its value is only partially defined during assembly and is completed at link time. An expression whose terms contain a global symbol not defined in the current program is an external expression. External expressions have relocatable values at execution time if the global symbol is defined as being relocatable or absolute if the global symbol is defined as absolute.

CHAPTER 4

RELOCATION AND LINKING

The output of the MACRO-11 Assembler is an object module which must be processed by Link-11 before loading and execution. (See PDP-11 Link-11 Linker and Libr-11 Librarian Programmer's Manual for details.) The Linker essentially fixes (i.e., makes absolute) the values of external or relocatable symbols and turns the object module into a load module.

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

All instructions that are to be modified (as described in the previous paragraph) are marked with an apostrophe in the assembly listing (see also Section 1.2). Thus the binary text output looks as follows:

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

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

ØØ5Ø65     CLR RELOCATABLE(5)    ;ASSUMING WE ARE IN THE ABSOLUTE
ØØØØ4Ø'                          ;SECTION AND THE VALUE OF
                                 ;RELOCATABLE IS RELOCATABLE 4Ø
```

CHAPTER 5

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 5.7, 5.8 and 5.11) 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:

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

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

   Examples:

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

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

d. 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.

5.1 REGISTER MODE

The register contains the operand.

```
Format for A:   R
Examples:       RØ=%Ø       ;DEFINE RØ AS REGISTER Ø
                CLR RØ      ;CLEAR REGISTER Ø
```

5.2 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
```

5.3 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 (RØ)+   ;EACH INSTRUCTION CLEARS
                CLR (RØ+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 JSR instructions using non-deferred autoincrement mode, autoincrement the register before its use on the PDP-11/2Ø (but not on the PDP-11/45 or 11/Ø5). 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, RØ originally contains 1ØØ.

```
MOV RØ,(Ø)+     ;THE QUANTITY 1Ø2 IS MOVED
                ;TO LOCATION 1ØØ

MOV RØ,-(Ø)     ;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/Ø5 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.

5.4 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 INCRE-
                          ;MENTED BY TWO
```

5.5 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 -(RØ)     ;DECREMENT CONTENTS OF REGISTERS
            CLR -(RØ+3)   ;Ø, 3, AND 2 BY TWO BEFORE USING
            CLR -(2)      ;AS ADDRESSES OF WORDS TO BE
                          ;CLEARED
```

5.6 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.
```

5.7 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.
```

5.8 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 1ØØ AND
                                ;LOCATION 114 HOLDS 2ØØØ,
                                ;LOCATION 2ØØØ IS CLEARED.
```

5.9 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 #1ØØ, RØ    ;MOVE AN OCTAL 1ØØ TO REGISTER Ø
                MOV #X, RØ      ;MOVE THE VALUE OF SYMBOL X TO
                                ;REGISTER Ø
```

The operation of this mode is explained as follows:

The statement MOV #1ØØ,R3 assembles as two words. These are:

```
Ø 1 2 7 Ø 3
Ø Ø Ø 1 Ø Ø
```

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.

5.10 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 @#1ØØ,RØ   ;MOVE THE VALUE OF THE CONTENTS
                              ;OF LOCATION 1ØØ TO REGISTER Ø.
               CLR @#X        ;CLEAR THE CONTENTS OF THE
                              ;LOCATION WHOSE ADDRESS IS X.
```

5.11 RELATIVE MODE

Relative mode is the normal mode for memory references.

```
Format for A: E
Examples:     CLR 1ØØ          ;CLEAR LOCATION 1ØØ.
              MOV X,Y          ;MOVE CONTENTS OF LOCATION X
                               ;TO LOCATION Y.
```

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

If the statement MOV 1ØØ,R3 is assembled at absolute location 2Ø, the assembled code is:

```
Location 2Ø:        Ø 1 6 7 Ø 3
Location 22:        Ø Ø Ø Ø 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=1ØØ, 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 1ØØ-.-4(PC),R3
```

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

5.12 RELATIVE DEFERRED MODE

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

Format for A: @E

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

5.13 TABLE OF MODE FORMS AND CODES

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

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

where n is the register number.

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

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

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

NOTE

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

The form @#E differs from the form E in that the second or third word of the instruction contains the absolute address of the operand rather than the relative distance between the operand and the PC. Thus, the instruction CLR @#1ØØ clears absolute location 1ØØ even if the instruction is moved from the point at which it was assembled. See the description of the .ENABLE AMA function in Section 6.2, which directs the assembly of all relative mode addresses as absolute mode addresses.

5.14 BRANCH INSTRUCTION ADDRESSING

The branch instructions are one 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:

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

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

c. 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.

NOTE

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

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.

CHAPTER 6

GENERAL ASSEMBLER DIRECTIVES

6.1 LISTING CONTROL DIRECTIVES

6.1.1 .LIST and .NLIST

Listing options can be specified in the text of a MACRO-11 program through the .LIST and .NLIST directives. These are of the form:

```
.LIST arg
.NLIST arg
```

where: arg represents one or more optional arguments.

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

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

     LTEST                      ;CALL THE MACRO

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

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

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

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

Allowable arguments for use with the listing directives are as follows (these arguments can be used singly or in combination:

Argument	Default	Function
SEQ	list	Controls the listing of source line sequence numbers. Error flags are normally printed on the line preceding the questionable source statement.
LOC	list	Controls the listing of the location counter (this field would not normally be suppressed).
BIN	list	Controls the listing of generated binary code.
BEX	list	Controls listing of binary extensions; that is, those locations and binary contents beyond the first binary word (per source statement). This is a subset of the BIN argument.
SRC	list	Controls the listing of the source code.
COM	list	Controls the listing of comments. This is a subset of the SRC argument and can be used to reduce listing time and/or space where comments are unnecessary.
MD	list	Controls listing of macro definitions and repeat range expansions.
MC	list	Controls listing of macro calls and repeat range expansions.

Argument	Default	Function
ME	no list	Controls listing of macro expansions.
MEB	no list	Controls listing of macro expansion binary code. A .LIST MEB causes only those macro expansion statements producing binary code to be listed. This is a subset of the ME argument.
CND	list	Controls the listing of unsatisfied conditions and all .IF and .ENDC statements. This argument permits conditional assemblies to be listed without including unsatisfied code.
LD	no list	Control listing of all listing directives having no arguments (those used to alter the listing level count).
TOC	list	Control listing of table of contents on pass 1 of the assembly (see Section 6.1.4 describing the .SBTTL directive). The full assembly listing is printed during pass 1 of the assembly.
TTM	Teletype mode	Controls listing output format. The TTM argument (the default case) causes output lines to be truncated to 72 characters. Binary code is printed with the binary extensions below the first binary word. The alternative (.NLIST TTM) to Teletype mode is line printer mode, which is shown in Figure 6-1.
SYM	list	Controls the listing of the symbol table for the assembly.

An example of an assembly listing as sent to a 132 column line printer is shown in Figure 6-1. Notice that binary extensions for statements generating more than one word are spread horizontally on the source line. An example of an assembly listing as sent to a teleprinter is shown in Figure 6-2. Notice that binary extensions for statements generating more than one word are printed on subsequent lines.

The listing options can also be specified through switches on the listing file specification in the command string to the MACRO-11 Assembler. These switches are:

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

```
 1 001766                                GETLIN: ;                       ;GET AN INPUT LINE
 2 001766                                        SAVREG
 3 001772  016700  000020'             1$:       MOV     FFCNT,R0      ;ANY RESERVED FF'S?
 4 001776  001420                                BEQ     31$           ;  NO
 5 002000  060067  000022'                       ADD     R0,PAGNUM     ;YES, UPDATE PAGE NUMBER
 6 002004  012767  177777  000026'               MOV     #-1,PAGEXT
 7 002012  005067  000012'                       CLR     LINNUM        ;INIT NEW CREF SEQUENCE
 8 002016  005067  000020'                       CLR     FFCNT
 9 002022  005067  000016'                       CLR     SEQEND
10 002026  005767  000000'                       TST     PASS
11 002032  001402                                BEQ     31$
12 002034  005067  000010'                       CLR     LPPCNT
13 002040  012702  001712'             31$:      MOV     #LINBUF,R2
14 002044  010267  000012'                       MOV     R2,LCBEGL     ;SEAT UP BEGINNING
15 002050  012767  002116' 000014'               MOV     #LINEND,LCENDL;  AND END OF LINE MARKERS
17 002056  005767  000200'                       TST     SMLCNT        ;IN SYSTEM MACRO?
18 002062  001145                                BNE     40$           ;  YES, SPECIAL
21 002064  016701  002214'                       MOV     MSBMRP,R1     ;ASSUME MACRO IN PROGRESS
22 002070  001166                                BNE     10$           ;BRANCH IF SO
24 002072  012701  000756'                       MOV     #SRCBUF,R1
25 002076                                       ,WAIT   #SRCLNK
26 002104  005267  000012'                       INC     LINNUM
27 002110  116700  000753'                       MOVB    SRCHDR+3,R0   ;GET CODE BYTE
28 002114  032700  000047                       BIT     #047,R0       ;ANYTHING BAD?
29 002120  001403                                BEQ     32$           ;  NO
30 002122                                       ERROR   L             ;YES, ERROR
31 002130  106100                        32$:    ROLB    R0            ;EOF?
32 002132  100014                                BPL     2$            ;  NO
33 002134  056767  000006' 000004'               BIS     CSISAV,ENDFLG
34 002142  001003                                BNE     34$
```

Example of MACRO-11 Line Printer Listing
(132 column line printer)

FIGURE 6-1

```
MACRO   V003A,1 24-MAY-72          MACRO V003A,1 25-MAY-72 00:31 PAGE 28
ASSEMBLER PROPER

1 001706                GETLIN:                                 ;GET AN INPUT LINE
2 001706                        SAVREG
3 001772 016700 1$:     MOV     FFCNT,R0                        ;ANY RESERVED FF'S?
         000020'
4 001776 001420         BEQ     3$                              ;  NO
5 002000 060067         ADD     R0,PAGNUM                       ;YES, UPDATE PAGE NUMBER
         000022'
6 002004 012767         MOV     #-1,PAGEXT
         177777
         000020'
7 002012 005067         CLR     LINNUM                          ;INIT NEW CREF SEQUENCE
         000012'
8 002016 005067         CLR     FFCNT
         000020'
9 002022 005067         CLR     SEQEND
         000010'
10 02026 005767         TST     PASS
         000000'
11 02032 001402         BEQ     31$
12 02034 005067         CLR     LPPCNT
         000016'
13 02040 012702 31$:    MOV     #LINBUF,R2
         001712'
14 02044 010267         MOV     R2,LCBEGL                       ;SEAT UP BEGINNING
         000012'
15 02050 012767         MOV     #LINEND,LCENDL                  ;  AND END OF LINE MARKERS
         002116'
         000014'
16                              .IF NDF XSML
17 02056 005767         TST     SMLCNT                          ;IN SYSTEM MACRO?
         000200'
18 02062 001145         BNE     40$                             ;  YES, SPECIAL
19                              .ENDC
20                              .IF NDF XMACRO
21 02064 016701         MOV     MSBMKP,R1                       ;ASSUME MACRO IN PROGRESS
         002214'
22 02070 001160         BNE     10$                             ;BRANCH IF SO
23                              .IFTF
24 02072 012701         MOV     #SRCBUF,R1
         000756'
25 02076                        .WAIT   #SRCLNK
26 02104 005267         INC     LINNUM
         000012'
27 02110 116700         MOVB    SRCBUF+3,R0                     ;GET CODE BYTE
         000753'
28 02114 032700         BIT     #047,R0                         ;ANYTHING BAD?
         000047
29 02120 001403         BEQ     32$                             ;  NO
30 02122                        ERROR   L                       ;YES, ERROR
31 02130 106100 32$:    ROLB    R0                              ;EOF?
32 02132 100014         BPL     2$                              ;  NO
33 02134 056767         BIS     CSISAV,ENDFLG
         000000'
         000004'
34 02142 001003         BNE     34$
```

Example of Page Heading from MACRO-11 Teletype Listing
(same format as for 80 column line printer).

FIGURE 6-2

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

NOTE

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

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

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

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

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

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

causes MACRO-11 to ignore all .LIST and .NLIST directives without arguments. This command string causes the listing of any source code which would have otherwise been suppressed. (The object file is sent to the system device; the source listing and symbol table are sent to the line printer.)

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

causes MACRO-11 to produce only an object file and a symbol table listing. The assembly listing is completely suppressed by the /NL switch. (The object file and symbol table file are sent to the system device.)

```
Line Seq.  Loc.      Binary Field
Nos.       Field    Binary Extension Field          Source Field                Comment Field

    1                                              .NLIST  TTM                 ;START OFF IN LP MODE
    2                                              .LIST   ME                  ;LIST MACRO EXPANSIONS
    3
    4                                              .MACRO  LSTMAC  ARG         ;LISTING ARGUMENT TEST
    5                                              .NLIST  ARG
    6                                              .WORD   1,2,3,4             ;COMMENT
    7                                              .LIST   ARG
    8                                              .ENDM
    9
   10 000000                                       LSTMAC  SEQ                 ;SEQENCE NUMBERS
                                                   .NLIST  SEQ
      000000  000001  000002  000003               .WORD   1,2,3,4             ;COMMENT
      000006  000004
                                                   .LIST   SEQ
   11
   12 000010                                       LSTMAC  LOC                 ;LOCATION COUNTER
                                          .NLIST   LOC
      000001  000002  000003              .WORD    1,2,3,4          ;COMMENT
      000004
                                                   .LIST   LOC
   13
   14 000020                                       LSTMAC  BIN                 ;BINARY
                      .NLIST  BIN
      000020          .WORD   1,2,3,4              ;COMMENT
                                                   .LIST   BIN
   15
   16 000030                                       LSTMAC  SRC                 ;SOURCE
      000030  000001  000002  000003
      000036  000004
                                                   .LIST   SRC
   17
   18 000040                                       LSTMAC  COM                 ;COMMENT
                                                   .NLIST  COM
      000040  000001  000002  000003               .WORD   1,2,3,4
      000046  000004
                                                   .LIST   COM
   19
   20 000050                                       LSTMAC  BEX                 ;BINARY EXTENSION LINES
                                                   .NLIST  BEX
      000050  000001  000002  000003               .WORD   1,2,3,4             ;COMMENT
```

```
Line
Seq.    Loc.                Binary Field
Nos.    Field               Binary Extension Field      Source Field                Comment Field

                                                       .LIST   BEX
        21
   22                         .LIST   TTM              ;TRY ABBREVIATED FORM
   23 00060 000001            .WORD   1,2,3,4          ;COMMENT
      00062 000002
      00064 000003
      00066 000004
   24
   25 00070                   LSTMAC  <COM,BEX>        ;COMBINATION TEST
                              .NLIST  COM,BEX
      00070 000001            .WORD   1,2,3,4
                              .LIST   COM,BEX
   26
        27                                             .NLIST  TTM             ;BACK TO EXPANDED LISTING
        28
        29          000001                             .END
```

6.1.2 Page Headings

The MACRO-11 Assembler outputs each page in the format shown in Figure 6-2 (Teletype listing). On the first line of each listing page the Assembler prints (from right to left):

a. title taken from .TITLE directive
b. assembler version identification
c. date
d. time-of-day
e. page number

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

6.1.3 .TITLE

The .TITLE directive is used to assign a name to the object module. The name is the first symbol following the directive and must be six Radix-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 object module of the assembled program to be named PROG (this name is distinguished from the filename of the object module specified in the command string to the Assembler). The name of the object module appears in the Linker load map and on the listing.

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

```
.MAIN.
```

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

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

6.1.4 .SBTTL

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

```
.SBTTL CONDITIONAL ASSEMBLIES
```

The text

```
CONDITIONAL ASSEMBLIES
```

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

During pass 1 of the assembly process, MACRO-11 automatically prints a table of contents for the listing containing the line sequence number and text of each .SBTTL directive in the program. Such a table of contents is inhibited by specifying the /NL:TOC switch option to the assembly listing file specification (or an .NLIST TOC directive within the source). For example:

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

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

An example of the table of contents is shown in Figure 6-3. Note that the first word of the subtitle heading is not limited to six characters since it is not a module name.

6.1.5 .IDENT

The .IDENT directive provides another means of labeling the object module produced as a result of a MACRO-11 assembly. In addition to the name assigned to the object module with the .TITLE directive, a character string (up to six characters, treated like a RAD5Ø string) can be specified between paired delimiters. For example:

```
.IDENT  /VØØ5A/
```

```
MACRO   V003A.1 24-MAY-72          MACRO V003A.1 25-MAY-72 00:31
TABLE OF CONTENTS

    5-  1          SECTOR INITIALIZATION
    7-  1          SUBROUTINE CALL DEFINITIONS
   12-  1          PARAMETERS
   14-  1          ROLL DEFINITIONS
   16-  1          PROGRAM INITIALIZATION
   26-  1          ASSEMBLER PROPER
   36-  1          STATEMENT PROCESSOR
   40-  1          ASSIGNMENT PROCESSOR
   41-  1          OP CODE PROCESSOR
   48-  1          EXPRESSION TO CODE-ROLL CONVERSIONS
   50-  1          CODE ROLL STORAGE
   51-  1          DIRECTIVES
   59-  1                  DATA-GENERATING DIRECTIVES
   68-  1          CONDITIONALS
   72-  1          LISTING CONTROL
   74-  1          ENABL/DSABL FUNCTIONS
   75-  1          CROSS REFERENCE HANDLERS
   78-  1          LISTING STUFF
   79-  1          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          PIN
```

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

Figure 6-3 Assembly Listing Table of Contents

The character string:

V$$5A

is converted to Radix-5$ notation and output to the global symbol directory of the object module.

This symbol can optionally be included in the load map listings output by the Linker.

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

6.1.6 Page Ejection

There are several means of obtaining a page eject in a MACRO-11 assembly listing:

a. After a line count of 58 lines, MACRO-11 automatically performs a page eject to skip over page perforations on line printer paper and to formulate Teletype output into pages.

b. 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.

c. 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.

6.2 FUNCTIONS: .ENABL AND .DSABL DIRECTIVES

Several functions are provided by MACRO-11 through the .ENABL and .DSABL directives. These directives use three-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 MACRO-11 language:

Symbolic Argument	Function
ABS	Enabling of this function produces absolute binary output; i.e., input to the Paper Tape Software System Absolute Loader (with a .BIN extension instead of .OBJ). The default case is .DSABL ABS; i.e., input to Link-11.
AMA	Enabling of this function directs the assembly of all relative addresses (address mode 67) as absolute addresses (address mode 37). This switch is useful during the debugging phase of program development.
CDR	The statement .ENABL CDR causes source columns 73 and greater to be treated as comment. This accommodates sequence numbers in card columns 72-80.
FPT	Enabling of this function causes floating point truncation, rather than rounding, as is otherwise performed. .DSABL FPT returns to floating point rounding mode.
LC	Enabling of this function causes the Assembler to accept lower case ASCII input instead of converting it to upper case.
LSB	Enable or disable a local symbol block. While a local symbol block is normally entered by encountering a new symbolic label or .CSECT directive, .ENABL LSB forces a local symbol block which is not terminated until a label or .CSECT directive following the .DSABL LSB statement is encountered. The default case is .DSABL LSB.
PNC	The statement .DSABL PNC inhibits binary output until an .ENABL PNC is encountered. The default case is .ENABL PNC.

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

Once a program has been written using these functions, or not using them, the functions can be controlled through switches specified in the command string to the MACRO-11 Assembler. These switches are:

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

where: arg is any of the arguments defined for the .ENABL and .DSABL directives.

Use of these switches overrides the enabling or disabling of all occurrences of that argument in the program.

6.3 DATA STORAGE DIRECTIVES

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

```
.BYTE
.WORD
'
"
.ASCII
.ASCIZ
.RAD5Ø
↑B
↑D
↑O
```

These facilities are explained in the following Sections.

6.3.1 .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 SUC-
                       ;CESSIVE BYTES.
```

where a legal expression must have an absolute value (or contain a reference to an external symbol) and must result in 8 bits or less of data. The 16-bit value of the expression must have a high-order byte (which is truncated) that is either all zeros or all ones. Each operand expression is stored in a byte of the object program. Multiple operands are separated by commas and stored in successive bytes. For example:

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

If the high-order byte of the expression equates to a value other than Ø or -1, it is truncated to the low-order 8 bits and flagged with a T error code. If the expression is relocatable, an A-type warning flag is given.

At link time it is likely that relocation will result in an expression of more than 8 bits, in which case, the Linker prints an error code. For example:

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

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

In the case where X is defined in another program:

```
        .GLOBL X
        .BYTE X           ;PRODUCES A "W" FLAG
                          ;SINCE THE STATEMENT IS NOT
                          ;ACCEPTABLE IF X IS A LABEL.
```

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

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

6.3.2 .WORD

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

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

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

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

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

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

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

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

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

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

Note that the default .WORD directive occurs whenever there is a leading arithmetic or logical operator, or whenever a leading symbol is encountered 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
```

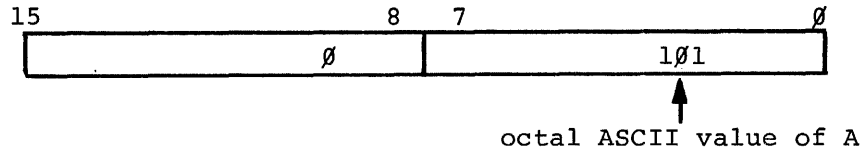
Two error codes result: Q occurs because an expression operator is missing between MOR and A, and a U occurs if MOR is undefined. Two words are then generated: one for MOR A and one for B.

6.3.3 ASCII Conversion of One or Two Characters

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

```
MOV #'A,RØ
```

results in the following 16 bits being moved into RØ:



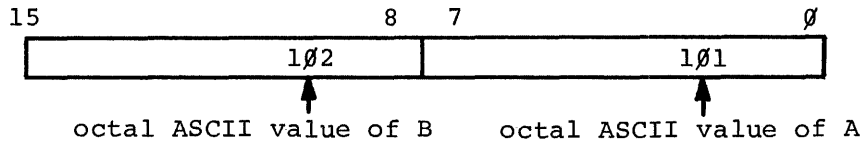
The ' character is never followed by a carriage return, null, rubout, line feed or form feed. (For another use of the ' character, see Section 7.3.6.)

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

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

```
MOV #"AB,RØ
```

results in the following word being moved into RØ:



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

```
;DEVICE NAME TABLE

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

6.3.4 .ASCII

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

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

where: character string — is a string of any acceptable printing ASCII characters. The string may not include null (blank) characters, rubout, 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 /ABC/<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 preempted by their significance as a comment indicator and assignment operator, respectively. For other than the first group, semicolons are treated as beginning a comment field. For example:

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

6.3.5 .ASCIZ

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

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

```
            :
            :
        MOV   #HELLO,R1
        MOV   #LINBUF,R2
X:      MOVB  (R1)+,(R2)+
        BNE   X
            :
            :
HELLO:  .ASCIZ  <CR><LF>/MACRO-11 VØØ1A/<CR><LF>   ;INTRO MESSAGE
                              :
                              :
```

6.3.6 .RAD5Ø

The .RAD5Ø 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:

```
        .RAD5Ø    /string/
```

where:

/ /	delimiters can be any printing characters other than the =, <, and ; characters.
string	is a list of the characters to be converted (three characters per word) and which may consist of the characters A through Z, Ø through 9, 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:

```
.RAD5Ø    /ABC          ;PACK ABC INTO ONE WORD.
.RAD5Ø    /AB/          ;PACK AB (SPACE) INTO ONE WORD.
.RAD5Ø    //            ;PACK 3 SPACES INTO ONE WORD.
.RAD5Ø    /ABCD/        ;PACK ABC INTO FIRST WORD AND
                        ;D SPACE SPACE INTO SECOND WORD.
```

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

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

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

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

Radix 50 value = ((C1*5Ø)+C2)*5Ø+C3

For example:

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

See Appendix A for a table to quickly determine Radix-50 equivalents.

Use of angle brackets is encouraged in the .ASCII, .ASCIZ, and .RAD5Ø statements whenever leaving the text string to insert special codes. For example:

```
.ASCII <1Ø1>      ;EQUIVALENT TO .ASCII/A/

.RAD5Ø /AB/<35>   ;STORES 3255 IN NEXT WORD

CHR1=1
CHR2=2
CHR3=3
   .
   .
   .
.RAD5Ø<CHR1><CHR2><CHR3> ;EQUIVALENT TO .RAD5Ø/ABC/
```

6.4 RADIX CONTROL

6.4.1 .RADIX

Numbers used in a MACRO-11 source program are initially considered to be octal numbers. However, the programmer has the option of declaring the following radices:

2, 4, 8, 1Ø

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 1Ø     ;BEGINS SECTION OF CODE WITH DECIMAL RADIX
   :
.RADIX        ;REVERTS TO OCTAL RADIX
```

In general it is recommended that macro definitions not contain or rely on radix settings from the .RADIX directive. The temporary radix control characters should be used within a macro definition. (↑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.

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

Once the user has specified a radix for a section of code, or has determined to use the default octal radix he may discover a number of cases where an alternate radix is more convenient

(particularly within macro definitions). For example, the creation of a mask word might best be done in the binary radix.

MACRO-11 has three unary operators to provide a single interpretation in a given radix within another radix as follows:

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

For example:

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

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

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

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

```
 1ØØ.    (144 base 8)
1376.    (254Ø base 8)
 128.    (2ØØ base 8)
```

6.5 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.

6.5.1 .EVEN

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

The .EVEN directive is used as follows:

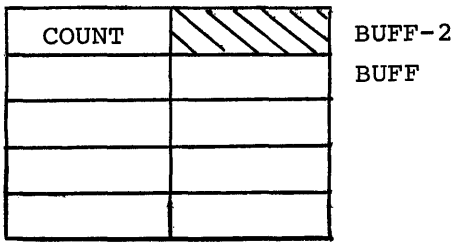
```
        .ASCIZ /THIS IS A TEST/
        .EVEN                   ;ASSURES NEXT STATEMENT
                                ;BEGINS ON A WORD BOUNDARY.
        .WORD XYZ
```

6.5.2 .ODD

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

```
;CODE TO MOVE DATA FROM AN INPUT LINE
;TO A BUFFER
        N=5                     ;BUFFER HAS 5 WORDS
          :
          :
        .ODD
        .BYTE   N*2             ;COUNT=2N BYTES
BUFF:   .BLKW   N               ;RESERVE BUFFER OF N WORDS
          :
          :
        MOV     #BUFF,R2        ;ADDRESS OF EMPTY BUFFER IN R2
        MOV     #LINE,R1        :ADDRESS OF INPUT LINE IS IN R1
        MOVB    -1(R2),R0       ;GET COUNT STORED IN BUFF-1 IN R0
AGAIN:  MOVB    (R1)+,(R2)+     ;MOVE BYTE FROM LINE INTO BUFFER
        BEQ     DONE            ;WAS NULL CHARACTER SEEN?
        DEC     R0              ;DECREMENT COUNT
        BNE     AGAIN           ;NOT = 0, GET NEXT CHARACTER
          :
          :
        CLRB    -(R2)           ;OUT OF ROOM IN BUFFER, CLEAR LAST
DONE:                           ;WORD
          :
          :
LINE:   .ASCIZ  /TEXT/
```

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



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

The .BLKB directive has the same effect as

```
.=.+exp
```

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

6.6 NUMERIC CONTROL

Several directives are available to provide software complements to the floating-point hardware on the PDP-11.

A floating-point number is represented by a string of decimal digits. The string (which can be a single digit in length) may optionally contain a decimal point, and may be followed by an optional exponent indicator; in the form of the letter E and a signed decimal exponent. The list of number representations below contains seven distinct, valid representations of the same floating-point number:

```
3
3.
3.Ø
3.ØEØ
3EØ
.3El
3ØØE-2
```

As can be quickly inferred, the list could be extended indefinitely (e.g., 3ØØØE-3, .Ø3E2, etc.). A leading plus sign is ignored (e.g., +3.Ø is considered to be 3.Ø). Leading minus signs complement the sign bit. No other operators are allowed (e.g., 3.Ø+N is illegal).

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

Floating-point numbers are normally rounded. That is, when a floating-point number exceeds the limits of the field in which it is to be stored, the high-order excess bit is added to the low-order retained bit. For example, if the number were to be stored in a 2-word field, but more than 32 bits were needed for its value, the highest bit carried out of the field would be added to the least significant position. In order to enable floating-point truncation, the .ENABL FPT directive is used and .DSABL FPT is used to return to floating-point rounding (see Section 6.2).

6.6.1 .FLT2 and .FLT4

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

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

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

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

The following code was assembled with the 4-word floating-point math package:

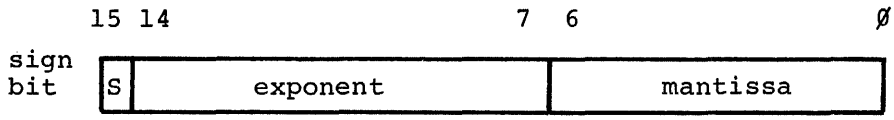
```
006010' 037314  146314  146314  ATOFTB: .FLT4   1.E-1           ;10↑-1
006016' 146315
006020' 036443  153412  036560          .FLT4   1.E-2           ;10↑-2
006026' 121727
006030' 034721  133427  054342          .FLT4   1.E-4           ;10↑-4
006036' 014545
006040' 031453  146167  010604          .FLT4   1.E-8           ;10↑-8
006046' 060717
006050' 022746  112624  137304          .FLT4   1.E-16          ;10↑-16
006056' 046741
006060' 005517  130436  126505          .FLT4   1.E-32          ;10↑-32
006066' 034625
```

6.6.2 Temporary Numeric Control: ↑F and ↑C

Like the temporary radix control operators, operators are available to specify either a one-word floating-point number (↑F) or the one's complement of a one-word number (↑C). For example:

```
FL3.7:    ↑F3.7
```

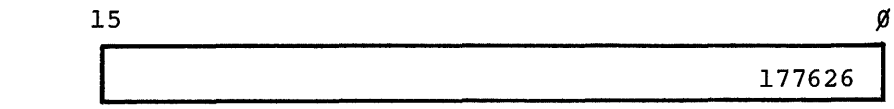
creates a one-word floating-point number at location FL3.7 containing the value 3.7 as follows:



This one-word floating-point number is the first word of the 2- or 4-word floating-point number format shown in the PDP-11 Processor Handbook, and the statement:

```
CMP151:    ↑C151
```

stores the one's complement of 151 in the current radix (assume current radix is octal) as follows:



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

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

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

```
↑C2+6
```

is equivalent to:

```
<↑C2>+6     or     177775+6     or     ØØØØØ3
```

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

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

```
 ↑Fl.Ø     =     Ø2Ø4ØØ
 ↑F-l.Ø    =     12Ø4ØØ

-↑Fl.Ø     =     1574ØØ
-↑F-l.Ø    =     Ø574ØØ
```

The argument to the ↑F operator must not be an expression and should be of the same format as arguments to the .FLT2 and .FLT4 directives (see Section 6.6.1).

6.7 TERMINATING DIRECTIVES

6.7.1 .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.

When the load module is loaded, program execution begins at the transfer address indicated by the .END exp directive. In a runtime system (the load module output of the Linker) an .END exp statement should terminate the first object module and .END statements should terminate any other object modules.

At the conclusion of the first assembly pass, upon encountering the END statement, MACRO-11 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 AØØ2 message is printed; the user is expected to reposition the tape in the reader and type CO (for CONTINUE).

6.7.2 .EOT

Under the Disk Operating System, the .EOT directive is ignored. The physical End-Of-Tape allows several physically separate tapes to be assembled as one program.

6.8 PROGRAM BOUNDARIES DIRECTIVE: .LIMIT

A program often wishes to know the boundaries of the load module's relocatable code. The .LIMIT directive reserves two words into which the Linker puts the low and high addresses of the relocated code. The low address (inserted into the first word) is the address of the first byte of code. The high address is the address of the first free byte following the relocated code. These addresses are always even since all relocatable sections are loaded at even addresses. (If a relocatable section consists of an odd number of bytes, the Linker adds one to the size to make it even.)

6.9 PROGRAM SECTION DIRECTIVES

```
.ASECT
.CSECT
.CSECT  symbol
```

The Assembler provides for 255_{10} program sections: an absolute section declared by .ASECT, an unnamed relocatable program section declared by .CSECT, and 253_{10} named relocatable program sections declared by .CSECT symbol, where symbol is any legal symbolic name. These directives allow the user to:

1. Create his program (object module) in sections:

   The Assembler maintains separate location counters for each section. This allows the user to write statements which are not physically contiguous but will be loaded contiguously. The following examples will clarify this:

```
     .CSECT             ;START THE UNNAMED RELOCATABLE SECTION
A:   Ø                  ;ASSEMBLED AT RELOCATABLE Ø,
B:   Ø                  ;      RELOCATABLE 2 AND
C:   Ø                  ;      RELOCATABLE 4,
ST:  CLR A              ;ASSEMBLE CODE AT
     CLR B              ;      RELOCATABLE ADDRESS
     CLR C              ;      6 THROUGH 21
     .ASECT             ;START THE ABSOLUTE SECTION
     .=4                ;ASSEMBLE CODE AT
     .WORD .+2,HALT     ;ABSOLUTE 4 THROUGH 7,
     .CSECT             ;RESUME THE UNNAMED RELOCATABLE
                        ;   SECTION
     INC A              ;ASSEMBLE CODE AT
     BR ST              ;   RELOCATABLE 22 THROUGH 27,
     .END
```

The first appearance of .CSECT or .ASECT assumes the location counter is at relocatable or absolute zero, respectively. The scope of each directive extends until a directive to the contrary is given. Further occurrences of the same .CSECT or .ASECT resume assembling where the section was left off.

```
        .CSECT  COM1        ;DECLARE SECTION COM1
A:      Ø                   ;ASSEMBLED AT RELOCATABLE Ø.
B:      Ø                   ;ASSEMBLED AT RELOCATABLE 2.
C:      Ø                   ;ASSEMBLED AT RELOCATABLE 4.
        .CSECT  COM2        ;DECLARE SECTION COM2
X:      Ø                   ;ASSEMBLED AT RELOCATABLE Ø.
Y:      Ø                   ;ASSEMBLED AT RELOCATABLE 2.
        .CSECT  COM1        ;RETURN TO COM1
D:      Ø                   ;ASSEMBLED AT RELOCATABLE 6.
        .END
```

The Assembler automatically begins assembling at relocatable zero of the unnamed .CSECT if not instructed otherwise; that is, the first statement of an assembly is an implied .CSECT.

All labels in an absolute section are absolute; all labels in a relocatable section are relocatable. The location counter symbol, ".", is relocatable or absolute when referenced in a relocatable or absolute section, respectively. Undefined internal symbols are assigned the value of relocatable or absolute zero in a relocatable or absolute section, respectively. Any labels appearing on a .ASECT or .CSECT statement are assigned the value of the location counter before the .ASECT or .CSECT takes effect. Thus, if the first statement of a program is:

```
A:      .ASECT
```

then A is assigned to relocatable zero and is associated with the unnamed relocatable section (because the Assembler implicitly begins assembly in the unnamed relocatable section).

Since it is not known at assembly time where the program sections are to be loaded, all references between sections in a single assembly are translated by the Assembler to references relative to the base of that section. The Assembler provides the Linker with the necessary information to resolve the linkage. Note that this is not necessary when making a reference to an absolute section (the Assembler knows all load addresses of an absolute section).

Examples:

```
     .ASECT
     .=1ØØØ
A:   CLR X          ;ASSEMBLED AS CLR BASE OF UNNAMED
                    ;     RELOCATABLE SECTION + 1Ø
     JMP Y          ;ASSEMBLED AS JMP BASE OF UNNAMED
                    ;     RELOCATABLE SECTION + 6
     .CSECT
     MOV RØ,R1
     JMP A          ;ASSEMBLED AS JMP 1ØØØ
Y:   HALT
X:   Ø
     .END
```

In the above example the references to X and Y were translated into references relative to the base of the unnamed relocatable section.

2. Share code and/or data between object modules (separate assembles):

Named relocatable program sections operate as FORTRAN labeled COMMON; that is, sections of the same name from different assemblies are all loaded at the same location by Link-11. The unnamed relocatable section is the exception to this as all unnamed relocatable sections are loaded in unique areas by Link-11.

Note that there is no conflict between internal symbolic names and program section names; that is, it is legal to use the same symbolic name for both purposes. In fact, considering FORTRAN again, this is a necessity to accommodate the FORTRAN statement:

```
COMMON  /X/A,B,C,X
```

where the symbol X represents the base of this program section and also the fourth element of this program section.

Also, there is no conflict between program section names and .GLOBL names. In FORTRAN language, COMMON block names and SUBROUTINE names may be the same.

6.10 SYMBOL CONTROL: .GLOBL

The Assembler produces a relocatable object module and a listing file containing the assembly listing and symbol table. Link-11 joins separately assembled object modules into a single load module. Object modules are relocated as a function of the specified base of the load module. The object modules (where there are more than one) are linked via common global symbols, such that a global symbol in one module (either defined by direct assignment or as a label) can be referenced from another module.

A global symbol must be specified in a .GLOBL directive. The form of the .GLOBL directive is:

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

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

Symbols appearing in a .GLOBL directive are either defined within the current program or are external symbols in which case they are defined in another program which is to be linked with the current program, by Link-11, prior to execution.

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

At the end of assembly pass 1, MACRO-11 has determined whether a given global symbol is defined within the program or is expected to be an external symbol. All internal symbols to a given program, then, must be defined by the end of pass 1.

```
;DEFINE A SUBROUTINE WITH 2 ENTRY POINTS WHICH CALLS AN
;         EXTERNAL SUBROUTINE
        .CSECT              ;DECLARE THE CONTROL SECTION
        .GLOBL  A,B,C       ;DECLARE A, B, C AS GLOBALS
A:      MOV     @(R5)+,RØ   ;ENTRY A DEFINED
        MOV     #X,R1
X:      JSR     PC,C        ;CALL EXTERNAL SUBROUTINE C
        RTS     R5          ;EXIT
B:      MOV     @(R5)+,R1   ;DEFINE ENTRY B
        CLR     R1
        BR      X
```

In the example on the previous page, A and B are entry symbols (entry points), C is an external symbol and X is an internal symbol.

A global symbol is defined only when it appears in a .GLOBL directive. A symbol is not considered a global symbol if it is assigned the value of a global expression in a direct assignment statement.

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

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

or a direct reference plus or minus a constant, i.e.:

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

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

6.11 CONDITIONAL ASSEMBLY DIRECTIVES

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

The general form of a conditional block is as follows:

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

where: cond is a condition which must be met if the block is to be included in the assembly. These conditions are defined below.

argument(s) are a function of the condition to be tested.

range is the body of code which is included in the assembly or ignored depending upon whether the condition is met.

The following are the allowable conditions:

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

NOTE

A macro-type argument is enclosed in angle brackets or within an up-arrow construction (as described in Section 7.3.1). For example:

```
<A,B,C>
↑/124/
```

For example:

```
.IF EQ    ALPHA+1       ;ASSEMBLE IF ALPHA+1=Ø
    :
    :
.ENDC
```

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

&	logical AND operator
!	logical inclusive OR operator

For example:

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

assembles if both SYM1 and SYM2 are defined.

6.11.1 Subconditionals

Subconditionals may be placed within conditional blocks to indicate:

a. 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.
b. assembly of a non-contiguous body of code within the conditional block depending upon the result of the conditional test to enter the block.
c. unconditional assembly of a body of code within a conditional block.

There are three subconditional directives, as follows:

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

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

For example:

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

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

However,

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

6.11.2 Immediate Conditionals

An immediate condiitonal directive is a means of writing a one-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 6.11.

arg — is the argument associated with the condition specified, that is, either an expression, symbol, or macro-type argument, as described in Section 6.11.

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
```

6.11.3 PAL-11R Conditional Assembly Directives

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

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

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

CHAPTER 7

MACRO DIRECTIVES

7.1 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.

7.1.1 .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.

7.1.2 .ENDM

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

```
.ENDM name
```

where:

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

For example:

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

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

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

An example of a macro definition is shown below:

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

7.1.3 .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=Ø, the .MEXIT directive terminates the macro expansion.

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

7.1.4 MACRO Definition Formatting

A form feed character used as a line terminator on a MACRO-11 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.

7.2 MACRO CALLS

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

```
label:  name, real arguments
```

where:

label	represents an optional statement label.
name	represents the name of the macro specified in the .MACRO directive preceding the macro definition.

	represents any legal separator (comma, space, or tab). No separator is necessary where there are no real arguments.
real arguments	are those symbols, expressions, and values 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 @RØ,R1        ;ABS IS USED AS 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.

7.3 ARGUMENTS TO MACRO CALLS AND DEFINITIONS

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

For example:

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

Arguments which contain separating characters are enclosed in paired angle brackets. An up-arrow construction is provided to allow angle brackets to be passed as arguments. Bracketed arguments are

seldom used in a macro definition, but are more likely in a macro call. For example:

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

This call would cause the entire statement:

```
MOV X,Y
```

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

The up-arrow construction could have been used in the above macro call as follows:

```
REN  ↑/MOV X,Y/,#44,WEV
```

which is equivalent to:

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

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

The form:

```
REN #44,WEV↑/MOV X,Y/
```

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

7.3.1 Macro Nesting

Macro nesting (nested macro calls), where the expansion of one macro includes a call to another macro, causes one set of angle brackets

to be removed from an argument with each nesting level. The depth of nesting allowed is dependent upon the amount of core space used by the program. To pass an argument containing legal argument delimiters to nested macros, the argument should be enclosed in one set of angle brackets for each level of nesting, as shown below:

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

.MACRO  LEVEL2  DUM3
DUM3
ADD #1Ø,RØ
MOV RØ, (R1)+
.ENDM
```

A call to the LEVEL1 macro:

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

causes the following expansion:

```
MOV X,RØ
ADD #1Ø,RØ
MOV RØ,(R1)+
CLR RØ
ADD #1Ø,RØ
MOV RØ,(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.

7.3.2 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)
```

7.3.3 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 function is defined in context. For example:

```
        B=Ø
        .MACRO  INC A,B
        CNT    A,\B
        B=B+1
        .ENDM
        .MACRO CNT  A,B
A'B:    .WORD
        .ENDM
          .
          .
          .
        INC  X,C
```

The macro call would expand to:

```
XØ:  .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 Ø, a single Ø character is passed to the macro expansion.)

The number being passed can also be used to make source listings somewhat clearer. For example, versions of programs created through conditional assembly of a single source can identify themselves as follows:

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

The above macro call expands to

```
.IDENT  /ØØ5AXX/
```

where XX is the conditional value of ID.

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

7.3.4 Number of Arguments

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

A macro can be defined with no arguments.

7.3.5 Automatically Created Symbols

MACRO-11 can be made to create symbols of the form n$ where n is a decimal integer number such that $64 \leq n \leq 127$. Created symbols are always local symbols between 64$ and 127$. (For a description of local symbols, see Section 3.5.) Such local symbols are created by the Assembler in numerical order, i.e.:

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

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

The range of these local symbols extends between two explicit labels. Each new explicit label causes a new local symbol block to be initialized.

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

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

Local symbols are generated only where the real argument of the macro call is either null or missing. If a real argument is specified in the macro call, the generation of a local symbol is inhibited and normal replacement is performed. Consider the following expansions of the macro ALPHA above.

GENERATE A LOCAL SYMBOL FOR MISSING ARGUMENT:

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

DO NOT GENERATE 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.

7.3.6 Concatenation

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

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

When this macro is called:

```
        DEF X,Y,<MACRO-11>
```

it expands as follows:

```
        XY:   .ASCIZ  /MACRO-11/
              .WORD  'X'Y
```

In the macro definition, the scan terminates upon finding the first ' character. Since A is a dummy argument, the ' is removed. The scan resumes with B, notes B as another dummy argument and concatenates the two dummy arguments. The third dummy argument is noted as going into the operand of the .ASCIZ directive. On the

next line (this is not a useful example, but one for purely 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.

7.4 .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 MACRO-11 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 is 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 7.3.

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

```
.MACRO SAVE     ARG
.NTYPE SYM,ARG
.IF    EQ,SYM&7Ø
MOV    ARG,TEMP         ;REGISTER MODE
.IFF
MOV    #ARG,TEMP        ;NON-REGISTER MODE
.ENDC
.ENDM
```

7.5 .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 MACRO-11 program, the Assembler outputs a single line containing:

a. the sequence number of the .ERROR directive line,
b. the current value of the location counter,
c. the value of the expression if one is specified, and,
d. the text string specified.

For example:

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

causes a line similar to the following to be output:

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

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 7Ø) 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.

7.6 INDEFINITE REPEAT BLOCK: .IRP AND .IRPC

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

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

where: label is an optional statement label. A label may not appear on any .IRP statement within another macro definition, repeat range or indefinite repeat range, or on any .ENDM statement.

arg is a dummy argument which is successively replaced with the real arguments in the .IRP statement.

<real argument> is a list of arguments to be used in the expansion of the indefinite repeat range and enclosed in angle-brackets. Each real argument is a string of zero or more characters or a list of real arguments (enclosed in angle brackets). The real arguments are separated by commas.

range is the block of code to be repeated once for each real argument in the list. The range may contain macro definitions, repeat ranges, or other indefinite repeat ranges. Note that only created symbols should be used as labels within an indefinite repeat range.

An indefinite repeat block can occur either within or outside macro definitions, repeat ranges, or indefinite repeat ranges. The rules for creating an indefinite repeat block are the same as for the creation of a macro definition (for example, the .MEXIT statement is allowed in an indefinite repeat block). Indefinite repeat arguments follow the same rules as macro arguments.

```
1                                       .TITLE  IRPTST
2                                       .LIST   MD,ME
3                                       .MCALL  .PARAM
4  000000                               .PARAM
           000000 R0=%^D00
           000001 R1=%^D01
           000002 R2=%^D02
           000003 R3=%^D03
           000004 R4=%^D04
           000005 R5=%^D05
           000006 R6=%^D06
           000007 R7=%^D07
           000006 SP=%^D06
           000007 PC=%^D07
           177776 PSW=^O177776
           177570 SWR=^O177570
5  000000 012700                        MOV     #TABLE,R0
           000056'
6
7                                       .IRP    X,<A,B,C,D,E,F>
8
9                                       MOV     X,(R0)+
10
11                                      .ENDM

     00004 016720                       MOV     A,(R0)+
           000032

     00010 016720                       MOV     B,(R0)+
           000030

     00014 016720                       MOV     C,(R0)+
           000026

     00020 016720                       MOV     D,(R0)+
           000024

     00024 016720                       MOV     E,(R0)+
           000022

     00030 016720                       MOV     F,(R0)+
           000020
```

Figure 7-1

.IRP and .IRPC Example

```
12
13                              .IRPC   X,ABCDEF
14
15                              .ASCII  /X/
16
17                              .ENDM

    000034      101             .ASCII  /A/

    000035      102             .ASCII  /B/

    000036      103             .ASCII  /C/

    000037      104             .ASCII  /D/

    000040      105             .ASCII  /E/

    000041      106             .ASCII  /F/

18
19
20  000042 041101 A:            .WORD   "AB
21  000044 041502 B:            .WORD   "BC
22  000046 042103 C:            .WORD   "CD
23  000050 042504 D:            .WORD   "DE
24  000052 043105 E:            .WORD   "EF
25  000054 043506 F:            .WORD   "FG
26  000056         TABLE:       .BLKW   6
27
28          000001              .END
```

Figure 7-1, Continued

.IRP and .IRPC Example

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

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

On each iteration of the indefinite repeat range, the dummy argument (arg) assumes the value of each successive character in the string. Terminators for the string are: space, comma, tab, carriage return, line feed, and semi-colon.

7.7 REPEAT BLOCK: .REPT

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

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

where:

- label — is an optional statement label. The .ENDR or .ENDM directive may not have a label. A .REPT statement occurring within another repeat block, indefinite repeat block, or macro definition may not have a label associated with it.
- expr — is any legal expression controlling the number of times the block of code is assembled. Where expr≤Ø, the range of the repeat block is not assembled.
- range — is the block of code to be repeated expr number of times. The range may contain macro definitions, 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.

7.8 MACRO LIBRARIES: .MCALL

All macro definitions must occur prior to their referencing within the user program. MACRO-11 provides a selection mechanism for the programmer to indicate in advance those system macro definitions required by his program.

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

```
.MCALL arg1,arg2,...
```

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

When this directive is encountered, MACRO-11 searches the system library files to find the requested definition(s). The system library file, SYSMAC.SML, is first sought under the current user's UIC on the system device where, if found, the Assembler takes the definition for all requested macros. If all macro requests have not been satisfied, or if SYSMAC.SML does not exist under the current UIC, the Assembler searches for the file SYSMAC:SML under [1,1] on the system device.

See Appendix D for a listing of the system macro file (SYSMAC.SML) stored under [1,1] on the system device.

CHAPTER 8

OPERATING PROCEDURES

The MACRO-11 Assembler assembles one or more ASCII source files containing MACRO-11 statements into a single relocatable binary object file. The output of the Assembler consists of a binary object file and an assembly listing followed by the symbol table listing. A CREF (cross reference) listing can be specified as part of the assembly output by means of a switch option.

8.1 LOADING MACRO-11

MACRO-11 is loaded with the Disk Monitor RUN command as follows:

```
$RUN MACRO
```

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

```
MACRO  VØØ1A
#
```

8.2 COMMAND INPUT STRING

In response to the # printed by the Assembler, the user types the output file specification(s), followed by a left angle bracket, followed by the input file specification(s):

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

where:

object	is the binary object file
listing	is the assembly listing file containing the assembly listing and symbol table and, optionally, a separate CREF listing file can be appended to the assembly listing or output as a separate file.
sourcel,source2, ...,sourceN	are the ASCII source files containing the MACRO-11 source program(s). No limit is set on the number of source input files, except as the Assembler is limited by the size of the user-defined and macro symbol tables.

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

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

One or more switch options can be specified with each file specification to provide the Assembler with information about that file. The switch options are described in Section 8.3.

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

Error	Error Code
Illegal switch Too many switches Illegal switch value Too many switch values	S203
Too many output file specifications	S204
Input file missing	S206

The default value for each file specification is noted below:

	dev	filnam	ext	uic
object	system device	last source file name	.OBJ	current
listing	device used for object output	last source file name	.LST	current
CREF intermediate	system device	last source file name	.CRF	current
sourcel	system device	-	.MAC .PAL .null	current

	dev	filnam	ext	uic
source2 : : sourceN	device used for source1 (last source file specified)	-	.MAC .PAL .null	current
system macro file	system device	SYSMAC	.SML	current [1,1]

8.3 SWITCH OPTIONS

There are four types of switch options: listing options, functions, CREF specifications, and pass assembly controls. The listing options are described in detail in Section 6.1.1. The function options are described in detail in Section 6.2. Rather than repeat this information here, the reader is advised to turn to these sections or the summary contained in Appendix B. The switch options are specified in the form:

Specification	Function
/LI /LI:arg /NL: /NL:arg	Listing control
/EN:arg /DS:arg	Function Control
/CRF /CRF:arg	Produce cross reference table
/PA:1	Assemble file during Pass 1 only
/PA:2	Assemble file during Pass 2 only

Switch options specified on the output side apply to both the object and listing files. Switch options specified on the input side apply to the particular file which the switch follows and all subsequent files.

8.4 CREF, CROSS-REFERENCE TABLE GENERATION

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

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

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

This command string sends the assembly listing (FILE2.LST) to the line printer. An intermediate CREF file is created

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

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

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

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

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

```
$RU CREF
CREF VØØ1A
#LP:<FILE2.CRF
```

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

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

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

The CREF listing is organized into one to five sections, each listing a different type of symbol. The sections are as follows:

Section Type	Argument
user-defined symbols	:S
macro symbolic names	:M
permanent symbols (instructions, directives)	:P

Section Type	Argument
.CSECT symbolic names	:C
error codes	:E

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

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

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

Figure 8-1 contains a segment of source code and Figure 8-2 contains a segment of a CREF listing with some references to the code in Figure 8-1. Notice the appearance of the @ and # characters in the CREF listing. An @ character appears in the CREF listing wherever a destructive reference has been made to that symbol (i.e., the contents of that symbol have been altered at that point). A # character appears in the CREF listing wherever a symbol is defined.

```
MACRO   V001    17-APR-72       MACRO V000A1  17-APR-72 19:09 PAGE 72
OBJECT CODE HANDLERS

1                               .SBTTL  OBJECT CODE HANDLERS
2
3  012026               ENDP:                           ;END OF PASS HANDLER
4  012026                       CALL    SETMAX
   012026 004767               JSR     PC,SETMAX
          174240
5  012032 005767               TST     PASS            ;PASS ONE?
          000000'
6  012036 001142               BNE     ENDP2           ;BRANCH IF PASS 2
7  012040                       ENTOVR  4
8  012040 005767               TST     OBJLNK          ;PASS ONE, ANY OBJECT?
          001416'
9  012044 001517               BEQ     30$             ;  NO
10 12046 012767                MOV     #BLKT01,BLKTYP  ;SET BLOCK TYP1 1
          000001
          000542'
11 12054                       CALL    OBJINI          ;INIT THE POINTERS
   12054 004767                JSR     PC,OBJINI
          001542
12 12060 012701                MOV     #PRGTTL,R1      ;SET "FROM" INDEX
          000050'
13 12064 016702                MOV     RLOPNT,R2       ;  AND "TO" INDEX
          000540'
14 12070                       CALL    GSDDMP          ;OUTPUT GSD BLOCK
   12070 004767                JSR     PC,GSDDMP
          000660
15 12074 005046                CLR     -(SP)           ;INIT FOR SECTOR SCAN
16 12076 012667 10$:           MOV     (SP)+,ROLUPD    ;SET SCAN MARKER
          000006'
17 12102                       NEXT    SECROL          ;GET THE NEXT SECTOR
   12102 012700                MOV     #SECROL,R0
          000010
   12106 004767                JSR     PC,NEXT
          005400
18 12112 001450                BEQ     20$             ;BRANCH IF THROUGH
19 12114 016746                MOV     ROLUPD,-(SP)    ;SAVE MARKER
          000006'
20 12120 012701                MOV     #MODE,R1
          000006'
21 12124 011105                MOV     (R1),R5         ;SAVE SECTOR
22 12126 042705                BIC     #377,R5         ;ISOLATE IT
          000377
23 12132 000305                SWAB    R5              ;  AND PLACE IN RIGHT
24 12134 042711                BIC     #-1-<RELFLG>,(R1)       ;CLEAR ALL BUT REL BIT
          177737
25 12140 052721                BIS     #<GSDT01>+DEFFLG,(R1)+  ;SET TO TYPE 1, DEFINED
          000410
26 12144 010521                MOV     R5,(R1)+        ;ASSUME ABS
27 12146 001401                BEQ     11$             ;  OOPS!
28 12150 011141                MOV     (R1),-(R1)      ;  REL, SET MAX
29 12152 005067 11$:           CLR     ROLUPD          ;SET FOR INNER SCAN
          000006'
30 12156 012701 12$:           MOV     #SYMBOL,R1
          000002'
31 12162                       CALL    GSDDMP          ;OUTPUT THIS BLOCK
   12162 004767                JSR     PC,GSDDMP
          000566
```

Figure 8-1

Assembly Listing

```
MACRO   V001    17-APR-72       MACRO V000A1  17-APR-72 19:09 PAGE 72+
OBJECT CODE HANDLERS

32 12166                13$:    NEXT    SYMROL          ;FETCH THE NEXT SYMBOL
   12166 012700                 MOV     #SYMROL,R0
         000000
   12172 004767                 JSR     PC,NEXT
         005314
33 12176 001737                 BEQ     10$             ;  FINISHED WITH THIS GUY
34 12200 032767                 BIT     #GLBFLG,MODE    ;GLOBAL?
         000100
         000046'
35 12206 001767                 BEQ     13$             ;  NO
36 12210 126705                 CMPB    SECTOR,R5       ;YES, PROPER SECTOR?
         000007'
37 12214 001364                 BNE     13$             ;  NO
38 12216 042767                 BIC     #-1-<DEFFLG!RELFLG!GLBFLG>,MODE ;CLEAR MOST
         177627
         000006'
39 12224 052767                 BIS     #GSDT04,MODE    ;SET TYPE 4
         002000
         000006'
40 12232 000751                 BR      12$             ;OUTPUT IT
```

Figure 8-1 (Cont.) Assembly Listing

```
ENDMAC    27-44    109-33#
ENDP      23-23     72- 3#
ENDP1M    73-16     73-22#
ENDP2     72- 6     74- 1#
   .
   .
   .
MDEFLG    12- 7#    35-28     92- 8     92-24
MEXIT    116- 1#   116-41#
MODE      14- 6#    22-29@    34-12     35-17@    36-12     37- 4     40-43@
          45- 6@    48-16@    58-38@    64-23     70-10     72-20     72-34
          72-38@    72-39@    74-34     75-37     86- 8     91-20@   106-27
         110-34@
MOVBYT    16- 5     18- 9     28-44     74-41     83-11     83-20    108-19#
MPOP     109-42    121-17#
MPUSH    109-26    110-43    121- 1#
MSGARG    27- 9    121-18    121-40#
MSGBLK   121- 4    121-28    121-36#
MSGCNT    27-15    109-33    116- 6    121-41#
MSGEND   121- 9    121-28    121-43#
MSGMRP    25-19     27-25@   110-49@   121-42#
```

Figure 8-2

Excerpts from CREF Listing to Accompany Figure 8-1. Note particularly the CREF references for ENDP, ENDP2, and MODE.

APPENDIX A

MACRO-11 CHARACTER SETS

A.1 ASCII CHARACTER SET

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
Ø	ØØØ	NUL	NULL, TAPE FEED, CONTROL/SHIFT/P.
1	ØØ1	SOH	START OF HEADING: ALSO SOM, START OF MESSAGE, CONTROL/A.
1	ØØ2	STX	START OF TEXT; ALSO EOA, END OF ADDRESS, CONTROL/B.
Ø	ØØ3	ETX	END OF TEXT; ALSO EOM, END OF MESSAGE, CONTROL/C.
1	ØØ4	EOT	END OF TRANSMISSION (END); SHUTS OFF TWX MACHINES, CONTROL/D.
Ø	ØØ5	ENQ	ENQUIRY (ENQRY); ALSO WRU, CONTROL/E.
Ø	ØØ6	ACK	ACKNOWLEDGE; ALSO RU, CONTROL/F.
1	ØØ7	BEL	RINGS THE BELL. CONTROL/G.
1	Ø1Ø	BS	BACKSPACE; ALSO FEO, FORMAT EFFECTOR. BACKSPACES SOME MACHINES, CONTROL/H.
Ø	Ø11	HT	HORIZONTAL TAB. CONTROL/I.
Ø	Ø12	LF	LINE FEED OR LINE SPACE (NEW LINE); ADVANCES PAPER TO NEXT LINE, DUPLICATED BY CONTROL/J.
1	Ø13	VT	VERTICAL TAB (VTAB). CONTROL/K.
Ø	Ø14	FF	FORM FEED TO TOP OF NEXT PAGE (PAGE). CONTROL/L.
1	Ø15	CR	CARRIAGE RETURN TO BEGINNING OF LINE. DUPLICATED BY CONTROL/M.
1	Ø16	SO	SHIFT OUT; CHANGES RIBBON COLOR TO RED. CONTROL/N.
Ø	Ø17	SI	SHIFT IN; CHANGES RIBBON COLOR TO BLACK. CONTROL/O.
1	Ø2Ø	DLE	DATA LINK ESCAPE. CONTROL/B (DCØ).
Ø	Ø21	DC1	DEVICE CONTROL 1, TURNS TRANSMITTER (READER) ON, CONTROL/Q (X ON).
Ø	Ø22	DC2	DEVICE CONTROL 2, TURNS PUNCH OR AUXILIARY ON. CONTROL/R (TAPE, AUX ON).
1	Ø23	DC3	DEVICE CONTROL 3, TURNS TRANSMITTER (READER) OFF, CONTROL/S (X OFF).
Ø	Ø24	DC4	DEVICE CONTROL 4, TURNS PUNCH OR AUXILIARY OFF. CONTROL/T (AUX OFF).
1	Ø25	NAK	NEGATIVE ACKNOWLEDGE; ALSO ERR, ERROR. CONTROL/U.
1	Ø26	SYN	SYNCHRONOUS FILE (SYNC). CONTROL/V.
Ø	Ø27	ETB	END OF TRANSMISSION BLOCK; ALSO LEM, LOGICAL END OF MEDIUM. CONTROL/W.

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
Ø	Ø3Ø	CAN	CANCEL (CANCL). CONTROL/X.
1	Ø31	EM	END OF MEDIUM. CONTROL/Y.
1	Ø32	SUB	SUBSTITUTE. CONTROL/Z.
Ø	Ø33	ESC	ESCAPE. CONTROL/SHIFT/K.
1	Ø34	FS	FILE SEPARATOR. CONTROL/SHIFT/L.
Ø	Ø35	GS	GROUP SEPARATOR. CONTROL/SHIFT/M.
Ø	Ø36	RS	RECORD SEPARATOR. CONTROL/SHIFT/N.
1	Ø37	US	UNIT SEPARATOR. CONTROL/SHIFT/O.
1	Ø4Ø	SP	SPACE.
Ø	Ø41	!	
Ø	Ø42	"	
1	Ø43	#	
Ø	Ø44	$	
1	Ø45	%	
1	Ø46	&	
Ø	Ø47	'	ACCENT ACUTE OR APOSTROPHE.
Ø	Ø5Ø	(	
1	Ø51	)	
1	Ø52	*	
Ø	Ø53	+	
1	Ø54	,	
Ø	Ø55	-	
Ø	Ø56	.	
1	Ø57	/	
Ø	Ø6Ø	Ø	
1	Ø61	1	
1	Ø62	2	
Ø	Ø63	3	
1	Ø64	4	
Ø	Ø65	5	
Ø	Ø66	6	
1	Ø67	7	
1	Ø7Ø	8	
Ø	Ø71	9	
Ø	Ø72	:	
1	Ø73	;	
Ø	Ø74	<	
1	Ø75	=	
1	Ø76	>	
Ø	Ø77	?	
1	1ØØ	@	
Ø	1Ø1	A	
Ø	1Ø2	B	
1	1Ø3	C	
Ø	1Ø4	D	
1	1Ø5	E	
1	1Ø6	F	
Ø	1Ø7	G	
Ø	11Ø	H	
1	111	I	
1	112	J	
Ø	113	K	
1	114	L	
Ø	115	M	
Ø	116	N	
1	117	O	

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
Ø	12Ø	P	
1	121	Q	
1	122	R	
Ø	123	S	
1	124	T	
Ø	125	U	
Ø	126	V	
1	127	W	
1	13Ø	X	
Ø	131	Y	
Ø	132	Z	
1	133	[	SHIFT/K.
Ø	134	\	SHIFT/L.
1	135	]	SHIFT/M.
1	136	↑	*
Ø	137	←	**
Ø	14Ø	`	ACCENT GRAVE .
1	141	a	
1	142	b	
Ø	143	c	
1	144	d	
Ø	145	e	
Ø	146	f	
1	147	g	
1	15Ø	h	
Ø	151	i	
Ø	152	j	
1	153	k	
Ø	154	l	
1	155	m	
1	156	n	
Ø	157	o	
1	16Ø	p	
Ø	161	q	
Ø	162	r	
1	163	s	
Ø	164	t	
1	165	u	
1	166	v	
Ø	167	w	
Ø	17Ø	x	
1	171	y	
1	172	z	
Ø	173	{	
1	174	|	
Ø	175	}	THIS CODE GENERATED BY ALT MODE.
Ø	176	~	THIS CODE GENERATED BY PREFIX KEY (IF PRESENT)
1	177	DEL	DELETE, RUB OUT.

* ↑ appears as ^ on some machines.

** ← appears as _ (underscore) on some machines.

A.2 RADIX-5Ø CHARACTER SET

Character	ASCII Octal Equivalent	Radix-5Ø Equivalent
space	4Ø	Ø
A-Z	1Ø1 - 132	1 - 32
$	44	33
.	56	34
unused		35
Ø-9	60 - 71	36 - 47

The maximum Radix-5Ø value is, thus,

$$47*5Ø^2 + 47*5Ø + 47 = 174777$$

The following table provides a convenient means of translating between the ASCII character set and its Radix-5Ø equivalents. For example, given the ASCII string X2B, the Radix-5Ø equivalent is (arithmetic is performed in octal):

```
      X = 113ØØØ
      2 = ØØ24ØØ
      B = ØØØØØ2
   X2B  = 1154Ø2
```

Single Char. or First Char.		Second Character		Third Character	
A	ØØ31ØØ	A	ØØØØ5Ø	A	ØØØØØ1
B	ØØ62ØØ	B	ØØØ12Ø	B	ØØØØØ2
C	Ø113ØØ	C	ØØØ17Ø	C	ØØØØØ3
D	Ø144ØØ	D	ØØØ24Ø	D	ØØØØØ4
E	Ø175ØØ	E	ØØØ31Ø	E	ØØØØØ5
F	Ø226ØØ	F	ØØØ36Ø	F	ØØØØØ6
G	Ø257ØØ	G	ØØØ43Ø	G	ØØØØØ7
H	Ø31ØØØ	H	ØØØ5ØØ	H	ØØØØ1Ø
I	Ø341ØØ	I	ØØØ55Ø	I	ØØØØ11
J	Ø372ØØ	J	ØØØ62Ø	J	ØØØØ12
K	Ø423ØØ	K	ØØØ67Ø	K	ØØØØ13
L	Ø454ØØ	L	ØØØ74Ø	L	ØØØØ14
M	Ø5Ø5ØØ	M	ØØ1Ø1Ø	M	ØØØØ15
N	Ø536ØØ	N	ØØ1Ø6Ø	N	ØØØØ16
O	Ø567ØØ	O	ØØ113Ø	O	ØØØØ17
P	Ø62ØØØ	P	ØØ12ØØ	P	ØØØØ2Ø
Q	Ø651ØØ	Q	ØØ125Ø	Q	ØØØØ21
R	Ø7Ø2ØØ	R	ØØ132Ø	R	ØØØØ22
S	Ø733ØØ	S	ØØ137Ø	S	ØØØØ23
T	Ø764ØØ	T	ØØ144Ø	T	ØØØØ24
U	1Ø15ØØ	U	ØØ151Ø	U	ØØØØ25
V	1Ø46ØØ	V	ØØ156Ø	V	ØØØØ26
W	1Ø77ØØ	W	ØØ163Ø	W	ØØØØ27
X	113ØØØ	X	ØØ17ØØ	X	ØØØØ3Ø
Y	1161ØØ	Y	ØØ175Ø	Y	ØØØØ31
Z	1212ØØ	Z	ØØ2Ø2Ø	Z	ØØØØ32
$	1243ØØ	$	ØØ2Ø7Ø	$	ØØØØ33
.	1274ØØ	.	ØØ214Ø	.	ØØØØ34
unused	1325ØØ	unused	ØØ221Ø	unused	ØØØØ35
Ø	1356ØØ	Ø	ØØ226Ø	Ø	ØØØØ36
1	14Ø7ØØ	1	ØØ233Ø	1	ØØØØ37
2	144ØØØ	2	ØØ24ØØ	2	ØØØØ4Ø
3	1471ØØ	3	ØØ245Ø	3	ØØØØ41
4	1522ØØ	4	ØØ252Ø	4	ØØØØ42
5	1553ØØ	5	ØØ257Ø	5	ØØØØ43
6	16Ø4ØØ	6	ØØ264Ø	6	ØØØØ44
7	1635ØØ	7	ØØ271Ø	7	ØØØØ45
8	1666ØØ	8	ØØ276Ø	8	ØØØØ46
9	1717ØØ	9	ØØ3Ø3Ø	9	ØØØØ47

APPENDIX B

MACRO-11 ASSEMBLY LANGUAGE AND ASSEMBLER

B.1 SPECIAL CHARACTERS

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

B.2 ADDRESS MODE SYNTAX

n is an integer between 0 and 7 representing a register. R is a register expression, E is an expression, ER is either a register expression or an expression in the range 0 to 7.

Format	Address Mode Name	Address Mode Number	Meaning
R	Register	0n	Register R contains the operand. R is a register expression.
@R or (ER)	Deferred Register	1n	Register R contains the operand address.
(ER)+	Autoincrement	2n	The contents of the register specified by ER are incremented after being used as the address of the operand.
@(ER)+	Deferred Autoincrement	3n	ER contains the pointer to the address of the operand. ER is incremented after use.
-(ER)	Autodecrement	4n	The contents of register ER are decremented before being used as the address of the operand.
@-(ER)	Deferred Autodecrement	5n	The contents of register ER are decremented before being used as the pointer to the address of the operand.
E(ER)	Index	6n	E plus the contents of the register specified, ER, is the address of the operand.
@E(ER)	Deferred Index	7n	E added to ER gives the pointer to the address of the operand.
#E	Immediate	27	E is the operand
@#E	Absolute	37	E is the address of the operand.
E	Relative	67	E is the address of the operand.
@E	Deferred Relative	77	E is the pointer to the address of the operand.

B.3 INSTRUCTIONS

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

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

OP	Instruction mnemonic
R	Register expression
E	Expression
ER	Register expression or expression $0 \leq ER \leq 7$
AC	Floating point register expression
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 operations are:

SE	Source Effective address
FSE	Floating Source Effective Address
DE	Destination Effective address
FDE	Floating Destination Effective Address
||	Absolute value of
()	Contents of
→	Becomes

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

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

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

*	Conditionally set
-	Not affected
0	Cleared
1	Set

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

Logical operations are represented by the following symbols:

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

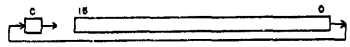
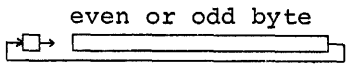
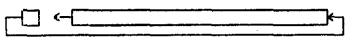
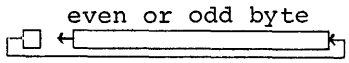
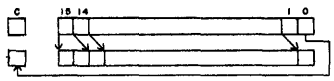
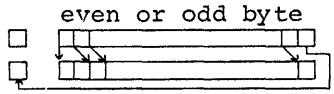
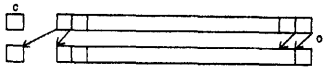
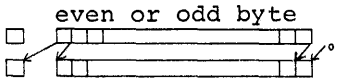
B.3.1 Double-Operand Instructions

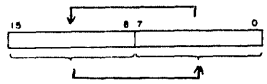
Instruction type format: Op A,A

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

B.3.2 Single-Operand Instructions

Instruction type format: Op A

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
0050DD 1050DD	CLR CLRB	CLear CLear Byte	Ø→DE	0	1	0	0
0051DD 1051DD	COM COMB	COMplement COMplement Byte	$(\overline{DE})$→DE	*	*	0	1
0052DD 1052DD	INC INCB	INCrement INCrement Byte	(DE)+1→DE	*	*	*	-
0053DD 1053DD	DEC DECB	DECrement DECrement Byte	(DE)-1→ DE	*	*	*	-
0054DD 1054DD	NEG NEGB	NEGate NEGate Byte	$(\overline{DE})$+1→ DE	*	*	*	*
0055DD 1055DD	ADC ADCB	ADd Carry ADd Carry Byte	(DE)+(C)→ DE	*	*	*	*
0056DD 1056DD	SBC SBCB	SuBtract Carry SuBtract Carry Byte	(DE)-(C)→ DE	*	*	*	*
0057DD 1057DD	TST TSTB	TeST TeST Byte	(DE)-Ø→ DE	*	*	0	0
0060DD	ROR	ROtate Right		*	*	*	*
1060DD	RORB	ROtate Right Byte		*	*	*	*
0061DD	ROL	ROtate Left		*	*	*	*
1061DD	ROLB	ROtate Left Byte		*	*	*	*
0062DD	ASR	Arithmetic Shift Right		*	*	*	*
1062DD	ASRB	Arithmetic Shift Right Byte		*	*	*	*
0063DD	ASL	Arithmetic Shift Left		*	*	*	*
1063DD	ASLB	Arithmetic Shift Left Byte		*	*	*	*

Op-Code	Mnemonic	Stands for	Operation	N	Z	V	C
0001DD	JMP	JuMP	DE→ PC	-	-	-	-
0003DD	SWAB	SWAp Bytes	□ 	*	*	0	0

The following instructions are available on the PDP-11/45 only:

0065DD	MFPI	Move From Previous Instruction space	See Chapter 6 in PDP-11/45 Processor Handbook	*	*	0	-
1065DD	MFPD	Move from Previous Data space		*	*	0	-
0066DD	MTPI	Move To Previous Instruction space		*	*	0	-
1066DD	MTPD	Move To Previous Data space		*	*	0	-
1701DD	LDFPS	Load FPP Program Status	(DE)→ FPS	-	-	-	-
0067DD	SXT	Sign eXTend	Ø→ DE if N bit clear -1→ DE if N bit is set	-	*	-	-
				FN	FZ	FV	FC
0707DD	NEGD	NEGate Double	-(FDE) → FDE	*	*	0	0
0707DD	NEGF	NEGate Floating	-(FDE) → FDE	*	*	0	0
1702DD	STFPS	STore Floating Point processor program Status	See Chapter 7 in PDP-11/45 Processor Handbook	-	-	-	-
1703DD	STST	STore floating point processor STatus		-	-	-	-
1704DD	CLRD	CLeaR Double	Ø→ FDE	0	1	0	0
1704DD	CLRF	CLeaR Floating	Ø→ FDE	0	1	0	0
1705DD	TSTD	TeST Double	(FDE)-Ø→FDE	*	*	0	0
1705DD	TSTF	TeST Floating	(FDE)-Ø→FDE	*	*	0	0

				FN	FZ	FV	FC
1706DD	ABSD	make ABSolute Double	|FDE|→FDE	0	*	0	0
1706DD	ABSF	make ABSolute Floating	|FDE|→FDE	0	*	0	0

B.3.3 Operate Instructions

Instruction Type format: Op

Op-Code	Mnemonic	Stands for	Operation	N	Z	V	C
000000	HALT	HALT	The computer stops all functions.	-	-	-	-
0000001	WAIT	WAIT	The computer stops and waits for an interrupt.	-	-	-	-
0000002	RTI	ReTurn from Interrupt	The PC and PS are popped off the SP stack: ((SP))→ PC (SP)+2→ SP ((SP))→ PS (SP)+2→ SP RTI is also used to return from a trap.	*	*	*	*
000005	RESET	RESET	Returns all I/O devices to power-on status.	-	-	-	-
000241	CLC	CLear Carry bit	Ø→ C	-	-	-	0
000261	SEC	SEt Carry bit	1→ C	-	-	-	1
000242	CLV	CLear oVerflow bit	Ø→ V	-	-	0	-
000262	SEV	SEt oVerflow bit	1→ V	-	-	1	-
000244	CLZ	CLear Zero bit	Ø→ Z	-	0	-	-
000264	SEZ	SEt Zero bit	1→ Z	-	1	-	-
000250	CLN	CLear Negative bit	Ø→ N	0	-	-	-
000270	SEN	SEt Negative bit	1→ N	1	-	-	-
000243		Clear OVerflow and Carry bits	Ø→ V Ø→ C	-	-	0	0
000254	CNZ	Clear Negative and Zero bits	Ø→ N Ø→ Z	0	0	-	-

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

The following instructions are available on the PDP-11/45 only:

Op-Code	Mnemonic	Stands for	Operation	FN	FZ	FV	FC
170000	CFCC	Copy Floating Condition Codes	Copy FPP condition codes into CPU condition codes.	*	*	*	*
000006	RTT	ReTurn from inTerrupt	Same as RTI instruction but inhibits trace trap	*	*	*	*
170011	SETD	SET Double floating mode	FPP set to double precision mode	-	-	-	-
170001	SETF	SET Floating mode	FPP set to single precision mode	-	-	-	-
170002	SETI	SET Integer mode	FPP set for integer data (16 bits)	-	-	-	-
170012	SETL	SET Long integer mode	FPP set for long integer data (32 bits)	-	-	-	-

B.3.4 Trap Instructions

Instruction type format: Op or Op E where $0 \leq E \leq 377_8$
*OP (only)

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

B.3.5 Branch Instructions

Instruction type format: Op E where $-128_{10} \leq (E-.-2)/2 \leq 127_{10}$

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

Op-Code	Mnemonic	Stands for	Condition to be met if branch is to occur
1030XX	BCC (or BHIS)	Branch if Carry Clear (or Branch if Higher or Same)	C=0
1034XX	BCS (or BLO)	Branch if Carry Set (or Branch if LOwer)	C=1

B.3.6 Register Destination

Instruction type format: OP ER,A

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

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

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

B.3.7 Register-Offset

Instruction type format: OP R,E

Op-Code	Mnemonic	Stands for	Operation	N	Z	V	C
077RDD	SOB	Subtract One and Branch	(R)-1→R PC-(2*DE)→ PC	-	-	-	-

B.3.8 Subroutine Return

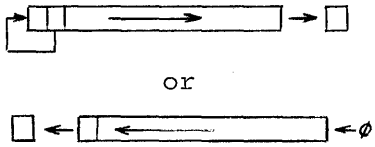
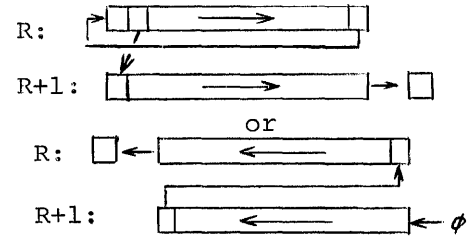
Instruction type format: Op ER

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

B.3.9 Source-Register

The following instructions are available on the PDP-11/45 only:

Instruction type format: Op A,R

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
071RSS	DIV	DIVide	R,Rv1/(SRC)→ R,Rv1	*	*	*	*
070RSS	MUL	MULtiply	R*(SRC) →R,Rv1	*	*	0	*
072RSS	ASH	Arithmetic SHift	R is shifted according to low-order 6-bits of source 	*	*	*	*
073RSS	ASHC	Arithmetic SHift Combined	R,Rv1 are shifted according to low-order 6 bits of source 	*	*	*	*

B.3.10 Floating-Point Source Double Register

The following instructions are available on the PDP-11/45 only:

Instruction type format: Op A,AC

Op-Code	Mnemonic	Stands for	Operation	Status Word Floating Condition Codes FN	FZ	FV	FC
172(AC)SS	ADDD	ADD Double	(FSE) +AC→ AC	*	*	*	0
172(AC)SS	ADDF	ADD Floating	(FSE) +AC→ AC	*	*	*	0
173(AC+4)SS	CMPD	CoMPare Double	(FSE) -AC	*	*	0	0
173(AC+4)SS	CMPF	CoMPare Floating	(FSE) -AC	*	*	0	0
174(AC+4)SS	DIVD	DIVide Double	AC/(FSE) → AC	*	*	*	0
174(AC+4)SS	DIVF	DIVide Floating	AC/(FSE) → AC	*	*	*	0
177(AC+4)SS	LDCDF	LoaD and Con-from Double to Floating	(FSE) → AC	*	*	*	0

Op-Code	Mnemonic	Stands for	Operation	Status Word Floating Condition Codes FN	FZ	FV	FC
177(AC+4)SS	LDCFD	LoaD and Convert from Floating to Double	(FSE) → AC	*	*	*	0
172(AC+4)SS	LDD	LoaD Double	(FSE) → AC	*	*	0	0
172(AC+4)SS	LDF	LoaD Floating	(FSE) → AC	*	*	0	0
171(AC+4)SS	MODD	Multiply and integerize double	AC*(FSE) → AC,AC1	*	*	*	0
171(AC+4)SS	MODF	Multiply and integerize floating-point	AC*(FSE) → AC	*	*	*	0
171(AC)SS	MULD	MULtiply Double	AC*(FSE) → AC	*	*	*	0
171(AC)SS	MULF	MULtiply Floating	AC*(FSE) → AC	*	*	*	0
173(AC)SS	SUBD	SUBtract Double	(FSE) -AC→ AC	*	*	*	0
173(AC)SS	SUBF	SUBtract Floating	(FSE) -AC→ AC	*	*	*	0

B.3.11 Source - Double Register

The following instructions are available on the PDP-11/45 only:

Instruction type format: Op A,AC

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes FN	FZ	FV	FC
177(AC)SS	LDCID	LoaD and Convert Integer to Double	(SE) → AC	*	*	*	0
177(AC)SS	LDCIF	LoaD and Convert Integer to Floating	(SE) → AC	*	*	*	0
177(AC)SS	LDCLD	LoaD and Convert Long integer to Double	(SE) → AC	*	*	*	0
177(AC)SS	LDCLF	LoaD and Convert Long Integer to Floating	(SE) → AC	*	*	*	0
176(AC+4)SS	LDEXP	LoaD EXPonent	(SE) +2ØØ/→ AC	*	*	0	0

B.3.12 Double Register - Destination

The following instructions are available on the PDP-11/45 only:

Instruction type format: Op AC,A

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes FN	FZ	FV	FC
176(AC)DD	STCFD	STore, Convert from Floating to Double	AC→ FDE	*	*	*	0
176(AC)DD	STCDF	STore, Convert from Double to Floating	AC→ FDE	*	*	*	0
175(AC+4)DD	STCDI[1]	STore, Convert from Double to Integer	AC→ FDE	*	*	0	*
175(AC+4)DD	STCDL[1]	STore, Convert from Double to Long integer	AC→ FDE	*	*	0	*
175(AC+4)DD	STCFI[1]	STore, Convert from Floating to Integer	AC→ FDE	*	*	0	*
175(AC+4)DD	STCFL[1]	STore, Convert from Floating to Long integer	AC→ FDE	*	*	0	*
174(AC)DD	STD	STore Double	AC→ FDE	-	-	-	-
174(AC)DD	STF	STore Floating	AC→ FDE	-	-	-	-
175(AC)DD	STEXP[1]	STore EXPonent	AC EXP-2ØØ→ DE	*	*	0	0

B.3.13 Number

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

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
0064NN	MARK	MARK	Stack cleanup on return from subroutine.	-	-	-	-

[1]These instructions set both the floating-point and processor condition codes as indicated.

B.3.14 Priority

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

Op-Code	Mnemonic	Stands for	Operation	Status Word Condition Codes N	Z	V	C
00023N	SPL	Set Priority Level	N→ PC(bits 7-5)	-	-	-	-

B.4 ASSEMBLER DIRECTIVES

Form	Operation	Described in Manual Section
'	A single quote character (apostrophe) followed by one ASCII character generates a word containing the 7-bit ASCII representation of the character in the low-order byte and zero in the high-order byte.	6.3.3
"	A double quote character followed by two ASCII characters generates a word containing the 7-bit ASCII representation of the two characters.	6.3.3
↑Bn	Temporary radix control; causes the number n to be treated as a binary number.	6.4.2
↑Cn	Creates a word containing the one's complement of n.	6.6.2
↑Dn	Temporary radix control; causes the number n to be treated as a decimal number.	6.4.2
↑Fn	Creates a one-word floating point quantity to represent n.	6.6.2
↑On	Temporary radix control; causes the number n to be treated as an octal number.	6.4.2
.ASCII string	Generates a block of data containing the ASCII equivalent of the character string(enclosed in delimiting characters) one character per byte.	6.3.4
.ASCIZ string	Generates a block of data containing the ASCII equivalent of the character string (enclosed in delimiting characters) one character per byte with a zero byte following the specified string.	6.3.5

Form	Operation	Described in Manual Section
.ASECT	Begin or resume absolute section.	6.9
.BLKB exp	Reserves a block of storage space exp bytes long.	6.5.3
.BLKW exp	Reserves a block of storage space exp words long.	6.5.3
.BYTE expl,exp2,...	Generates successive bytes of data containing the octal equivalent of the expression(s) specified.	6.3.1
.CSECT symbol .CSECT	Begin or resume named or unnamed relocatable section.	6.9
.DSABL arg	Disables the assembler function specified by the argument.	6.2
.ENABL arg	Provides the assembler function specified by the argument.	6.2
.END .END exp	Indicates the physical end of source program. An optional argument specifies the transfer address.	6.7.1
.ENDC	Indicates the end of a condition block.	6.11
.ENDM .ENDM symbol	Indicates the end of the current repeat block, indefinite repeat block, or macro. The optional symbol, if used, must be identical to the macro name.	7.1.2
.EOT	Ignored. Indicates End-of-Tape which is detected automatically by the hardware.	6.7.2
.ERROR exp,string	Causes a text string to be output to the command device containing the optional expression specified and the indicated text string.	7.5
.EVEN	Ensures that the assembly location counter contains an even address by adding 1 if it is odd.	6.5.1
.FLT2 argl,arg2,...	Generates successive two-word floating-point equivalents for the floating-point numbers specified as arguments.	6.6.1
.FLT4 argl,arg2,...	Generates successive four-word floating-point equivalents for the floating-point numbers specified as arguments.	6.6.1
.GLOBL syml,sym2,...	Defines the symbol(s) specified as global symbol(s).	6.10

Form	Operation	Described in Manual Section
.IDENT symbol	Provides a means of labeling the object module with the program version number. The symbol is the version number between paired delimiting characters.	6.1.5
.IF cond,arg1,arg2,...	Begins a conditional block of source code which is included in the assembly only if the stated condition is met with respect to the argument(s) specified.	6.11
.IFF	Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested false.	6.11.1
.IFT	Appears only within a conditional block and indicates the beginning of a section of code to be assembled if the condition tested true.	6.11.1
.IFTF	Appears only within a conditional block and indicates the beginning of a section of code to be unconditionally assembled.	6.11.1
.IIF cond,arg,statement	Acts as a one-line conditional block where the condition is tested for the argument specified. The statement is assembled only if the condition tests true.	6.11.2
.IRP sym,<arg1,arg2,...>	Indicates the beginning of an indefinite repeat block in which the symbol specified is replaced with successive elements of the real argument list (which is enclosed in angle brackets).	7.6
.IRPC sym,string	Indicates the beginning of an indefinite repeat block in which the symbol specified takes on the value of successive characters in the character string.	7.6
.LIMIT	Reserves two words into which the Linker inserts the low and high addresses of the relocated code.	6.8

Form	Operation	Described in Manual Section
.LIST .LIST arg	Without an argument, .LIST increments the listing level count by 1. With an argument, .LIST does not alter the listing level count but formats the assembly listing according to the argument specified.	6.1.1
.MACRO sym,arg1,arg2,...	Indicates the start of a macro named sym containing the dummy arguments specified.	7.1.1
.MEXIT	Causes an exit from the current macro or indefinite repeat block.	7.1.3
.NARG symbol	Appears only within a macro definition and equates the specified symbol to the number of arguments in the macro call currently being expanded.	7.4
.NCHR sym,string	Can appear anywhere in a source program; equates the symbol specified to the number of characters in the string (enclosed in delimiting characters).	7.4
.NLIST .NLIST arg	Without an argument, .NLIST decrements the listing level count by 1. With an argument, .NLIST deletes the portion of the listing indicated by the argument.	6.1.1
.NTYPE sym,arg	Appears only in a macro definition and equates the low-order six bits of the symbol specified to the six-bit addressing mode of the argument.	7.4
.ODD	Ensures that the assembly location counter contains an odd address by adding 1 if it is even.	6.5.1
.PAGE	Causes the assembly listing to skip to the top of the next page.	6.1.6
.PRINT exp,string	Causes a text string to be output to the command device containing the optional expression specified and the indicated text string.	7.5
.RADIX n	Alters the current program radix to n, where n can be 2, 4, 8, or 1Ø.	6.4.1

Form	Operation	Described in Manual Section
.RAD5Ø string	Generates a block of data containing the Radix-50 equivalent of the character string (enclosed in delimiting characters).	6.3.6
.REPT exp	Begins a repeat block. Causes the section of code up to the next .ENDM or .ENDR to be repeated exp times.	7.7
.SBTTL string	Causes the string to be printed as part of the assembly listing page header. The string part of each .SBTTL directive is collected into a table of contents at the beginning of the assembly listing.	6.1.4
.TITLE string	Assigns the first symbolic name in the string to the object module and causes the string to appear on each page of the assembly listing. One .TITLE directive should be issued per program.	6.1.3
.WORD expl,exp2,...	Generates successive words of data containing the octal equivalent of the expression(s) specified.	6.3.2

APPENDIX C

PERMANENT SYMBOL TABLE

```
PST     PERMANENT SYMBOL TABLE  MACRO V004A  PAGE 1


1                       .TITLE  PST     PERMANENT SYMBOL TABLE
2
3               ;       COPYRIGHT 1972  DIGITAL EQUIPMENT CORPORATION
4
5       000000'         .CSECT  PSTSEC
6
7                       .GLOBL  PSTBAS, PSTTOP          ;LIMITS
8                       .GLOBL  WRDSYM          ;POINTER TO .WORD
9
10      000200 DR1=     200                     ;DESTRUCTIVE REFERENCE IN FIRST
11      000100 DR2=     100                     ;DESTRUCTIVE REFERENCE IN SECOND
12
13                      .GLOBL  DFLGEV, DFLGBM, DFLCND, DFLMAC, DFLSMC
14
15      000020 DFLGEV=  020                     ;DIRECTIVE REQUIRES EVEN LOCATIO
16      000010 DFLGBM=  010                     ;DIRECTIVE USES BYTE MODE
17      000004 DFLCND=  004                     ;CONDITIONAL DIRECTIVE
18      000002 DFLMAC=  002                     ;MACRO DIRECTIVE
19      000001 DFLSMC=  001                     ;MCALL
20
21
22                      .IIF DF X45,    XFLTG=  0
23                      .IIF DF XMACRO, XSMCAL= 0
24
25                      .MACRO  OPCDEF  NAME,   CLASS,  VALUE,  FLAGS,  COND
26                      .IF NB  <COND>
27                      .IF DF  COND
28                      .MEXIT
29                      .ENDC
30                      .ENDC
31                      .RAD50  /NAME/
32                      .BYTE   FLAGS+0
33                      .GLOBL  OPCL'CLASS
34                      .BYTE   200+OPCL'CLASS
35                      .WORD   VALUE
36                      .ENDM
37
38                      .MACRO  DIRDEF  NAME,   FLAGS,  COND
39                      .IF NB  <COND>
40                      .IF DF  COND
41                      .MEXIT
42                      .ENDC
43                      .ENDC
44                      .GLOBL  NAME
45                      .RAD50  /.'NAME/
46                      .BYTE   FLAGS+0
47                      .BYTE   0
48                      .WORD   NAME
49                      .ENDM
50
51  00000       PSTBAS:                         ;BASE
```

```
1  000000                OPCDEF  <ABSD  >,      01,     170600, DR1,     X45
2  000010                OPCDEF  <ABSF  >,      01,     170600, DR1,     X45
3  000020                OPCDEF  <ADC   >,      01,     005500, DR1
4  000030                OPCDEF  <ADCB  >,      01,     105500, DR1
5  000040                OPCDEF  <ADD   >,      02,     060000, DR2
6  000050                OPCDEF  <ADDD  >,      11,     172000, DR2,     X45
7  000060                OPCDEF  <ADDF  >,      11,     172000, DR2,     X45
8  000070                OPCDEF  <ASH   >,      09,     072000, DR2,     X45
9  000100                OPCDEF  <ASHC  >,      09,     073000, DR2,     X45
10  00110                OPCDEF  <ASL   >,      01,     006300, DR1
11  00120                OPCDEF  <ASLB  >,      01,     106300, DR1
12  00130                OPCDEF  <ASR   >,      01,     006200, DR1
13  00140                OPCDEF  <ASRB  >,      01,     106200, DR1
14  00150                OPCDEF  <BCC   >,      04,     103000,
15  00160                OPCDEF  <BCS   >,      04,     103400,
16  00170                OPCDEF  <BEQ   >,      04,     001400,
17  00200                OPCDEF  <BGE   >,      04,     002000,
18  00210                OPCDEF  <BGT   >,      04,     003000,
19  00220                OPCDEF  <BHI   >,      04,     101000,
20  00230                OPCDEF  <BHIS  >,      04,     103000,
21  00240                OPCDEF  <BIC   >,      02,     040000, DR2
22  00250                OPCDEF  <BICB  >,      02,     140000, DR2
23  00260                OPCDEF  <BIS   >,      02,     050000, DR2
24  00270                OPCDEF  <BISB  >,      02,     150000, DR2
25  00300                OPCDEF  <BIT   >,      02,     030000,
26  00310                OPCDEF  <BITB  >,      02,     130000,
27  00320                OPCDEF  <BLE   >,      04,     003400,
28  00330                OPCDEF  <BLO   >,      04,     103400,
29  00340                OPCDEF  <BLOS  >,      04,     101400,
30  00350                OPCDEF  <BLT   >,      04,     002400,
31  00360                OPCDEF  <BMI   >,      04,     100400,
32  00370                OPCDEF  <BNE   >,      04,     001000,
33  00400                OPCDEF  <BPL   >,      04,     100000,
34  00410                OPCDEF  <BPT   >,      00,     000003,     ,    X45
35  00420                OPCDEF  <BR    >,      04,     000400,
36  00430                OPCDEF  <BVC   >,      04,     102000,
37  00440                OPCDEF  <BVS   >,      04,     102400,
38  00450                OPCDEF  <CCC   >,      00,     000257,
39  00460                OPCDEF  <CFCC  >,      00,     170000,     ,    X45
40  00470                OPCDEF  <CLC   >,      00,     000241,
41  00500                OPCDEF  <CLN   >,      00,     000250,
42  00510                OPCDEF  <CLR   >,      01,     005000, DR1
43  00520                OPCDEF  <CLRB  >,      01,     105000, DR1
44  00530                OPCDEF  <CLRD  >,      01,     170400, DR1,     X45
45  00540                OPCDEF  <CLRF  >,      01,     170400, DR1,     X45
46  00550                OPCDEF  <CLV   >,      00,     000242,
47  00560                OPCDEF  <CLZ   >,      00,     000244,
```

```
1  000570                OPCDEF  <CMP   >,      02,     020000,
2  000600                OPCDEF  <CMPB  >,      02,     120000,
3  000610                OPCDEF  <CMPD  >,      11,     173400,     ,     X45
4  000620                OPCDEF  <CMPF  >,      11,     173400,     ,     X45
5  000630                OPCDEF  <CNZ   >,      00,     000254,
6  000640                OPCDEF  <COM   >,      01,     005100, DR1
7  000650                OPCDEF  <COMB  >,      01,     105100, DR1
8  000660                OPCDEF  <DEC   >,      01,     005300, DR1
9  000670                OPCDEF  <DECB  >,      01,     105300, DR1
10 00700                 OPCDEF  <DIV   >,      07,     071000, DR2,    X45
11 00710                 OPCDEF  <DIVD  >,      11,     174400, DR2,    X45
12 00720                 OPCDEF  <DIVF  >,      11,     174400, DR2,    X45
13 00730                 OPCDEF  <EMT   >,      06,     104000,
14 00740                 OPCDEF  <HALT  >,      00,     000000,
15 00750                 OPCDEF  <INC   >,      01,     005200, DR1
16 00760                 OPCDEF  <INCB  >,      01,     105200, DR1
17 00770                 OPCDEF  <IOT   >,      00,     000004,
18 01000                 OPCDEF  <JMP   >,      01,     000100,
19 01010                 OPCDEF  <JSR   >,      05,     004000, DR1
20 01020                 OPCDEF  <LDCDF >,      11,     177400, DR2,    X45
21 01030                 OPCDEF  <LDCFD >,      11,     177400, DR2,    X45
22 01040                 OPCDEF  <LDCID >,      14,     177000, DR2,    X45
23 01050                 OPCDEF  <LDCIF >,      14,     177000, DR2,    X45
24 01060                 OPCDEF  <LDCLD >,      14,     177000, DR2,    X45
25 01070                 OPCDEF  <LDCLF >,      14,     177000, DR2,    X45
26 01100                 OPCDEF  <LDD   >,      11,     172400, DR2,    X45
27 01110                 OPCDEF  <LDEXP >,      14,     176400, DR2,    X45
28 01120                 OPCDEF  <LDF   >,      11,     172400, DR2,    X45
29 01130                 OPCDEF  <LDFPS >,      01,     170100,     ,     X45
30 01140                 OPCDEF  <LDSC  >,      00,     170004,     ,     X45
31 01150                 OPCDEF  <LDUB  >,      00,     170003,     ,     X45
32 01160                 OPCDEF  <MARK  >,      10,     006400,     ,     X45
33 01170                 OPCDEF  <MFPD  >,      01,     106500,     ,     X45
34 01200                 OPCDEF  <MFPI  >,      01,     006500,     ,     X45
35 01210                 OPCDEF  <MODD  >,      11,     171400, DR2,    X45
36 01220                 OPCDEF  <MODF  >,      11,     171400, DR2,    X45
37 01230                 OPCDEF  <MOV   >,      02,     010000, DR2
38 01240                 OPCDEF  <MOVB  >,      02,     110000, DR2
39 01250                 OPCDEF  <MTPD  >,      01,     106600, DR1,    X45
40 01260                 OPCDEF  <MTPI  >,      01,     006600, DR1,    X45
41 01270                 OPCDEF  <MUL   >,      07,     070000, DR2,    X45
42 01300                 OPCDEF  <MULD  >,      11,     171000, DR2,    X45
43 01310                 OPCDEF  <MULF  >,      11,     171000, DR2,    X45
44 01320                 OPCDEF  <NEG   >,      01,     005400, DR1
45 01330                 OPCDEF  <NEGB  >,      01,     105400, DR1
46 01340                 OPCDEF  <NEGD  >,      01,     170700, DR1,    X45
47 01350                 OPCDEF  <NEGF  >,      01,     170700, DR1,    X45
48 01360                 OPCDEF  <NOP   >,      00,     000240,
49 01370                 OPCDEF  <RESET >,      00,     000005,
```

```
1  001400                OPCDEF  <ROL   >,      01,     006100, DR1
2  001410                OPCDEF  <ROLB  >,      01,     106100, DR1
3  001420                OPCDEF  <ROR   >,      01,     006000, DR1
4  001430                OPCDEF  <RORB  >,      01,     106000, DR1
5  001440                OPCDEF  <RTI   >,      00,     000002,
6  001450                OPCDEF  <RTS   >,      03,     000200, DR1
7  001460                OPCDEF  <RTT   >,      00,     000006,    ,    X45
8  001470                OPCDEF  <SBC   >,      01,     005600, DR1
9  001500                OPCDEF  <SBCB  >,      01,     105600, DR1
10 01510                 OPCDEF  <SCC   >,      00,     000277,
11 01520                 OPCDEF  <SEC   >,      00,     000261,
12 01530                 OPCDEF  <SEN   >,      00,     000270,
13 01540                 OPCDEF  <SETD  >,      00,     170011,    ,    X45
14 01550                 OPCDEF  <SETF  >,      00,     170001,    ,    X45
15 01560                 OPCDEF  <SETI  >,      00,     170002,    ,    X45
16 01570                 OPCDEF  <SETL  >,      00,     170012,    ,    X45
17 01600                 OPCDEF  <SEV   >,      00,     000262,
18 01610                 OPCDEF  <SEZ   >,      00,     000264,
19 01620                 OPCDEF  <SOB   >,      08,     077000, DR1,    X45
20 01630                 OPCDEF  <SPL   >,      13,     000230,    ,    X45
21 01640                 OPCDEF  <STA0  >,      00,     170005,    ,    X45
22 01650                 OPCDEF  <STB0  >,      00,     170006,    ,    X45
23 01660                 OPCDEF  <STCDF >,      12,     176000, DR2,    X45
24 01670                 OPCDEF  <STCDI >,      12,     175400, DR2,    X45
25 01700                 OPCDEF  <STCDL >,      12,     175400, DR2,    X45
26 01710                 OPCDEF  <STCFD >,      12,     176000, DR2,    X45
27 01720                 OPCDEF  <STCFI >,      12,     175400, DR2,    X45
28 01730                 OPCDEF  <STCFL >,      12,     175400, DR2,    X45
29 01740                 OPCDEF  <STD   >,      12,     174000, DR2,    X45
30 01750                 OPCDEF  <STEXP >,      12,     175000, DR2,    X45
31 01760                 OPCDEF  <STF   >,      12,     174000, DR2,    X45
32 01770                 OPCDEF  <STFPS >,      01,     170200, DR1,    X45
33 02000                 OPCDEF  <STQ0  >,      00,     170007,    ,    X45
34 02010                 OPCDEF  <STST  >,      01,     170300, DR1,    X45
35 02020                 OPCDEF  <SUB   >,      02,     160000, DR2
36 02030                 OPCDEF  <SUBD  >,      11,     173000, DR2,    X45
37 02040                 OPCDEF  <SUBF  >,      11,     173000, DR2,    X45
38 02050                 OPCDEF  <SWAB  >,      01,     000300, DR1
39 02060                 OPCDEF  <SXT   >,      01,     006700, DR1,    X45
40 02070                 OPCDEF  <TRAP  >,      06,     104400,
41 02100                 OPCDEF  <TST   >,      01,     005700,
42 02110                 OPCDEF  <TSTB  >,      01,     105700,
43 02120                 OPCDEF  <TSTD  >,      01,     170500,    ,    X45
44 02130                 OPCDEF  <TSTF  >,      01,     170500,    ,    X45
45 02140                 OPCDEF  <WAIT  >,      00,     000001,
46 02150                 OPCDEF  <XOR   >,      05,     074000, DR2,    X45
```

```
1 002160                DIRDEF  <ASCII>,        DFLGBM
2 002170                DIRDEF  <ASCIZ>,        DFLGBM
3 002200                DIRDEF  <ASECT>
4 002210                DIRDEF  <BLKB >
5 002220                DIRDEF  <BLKW >,        DFLGEV
6 002230                DIRDEF  <BYTE >,        DFLGBM
7 002240                DIRDEF  <CSECT>
8 002250                DIRDEF  <DSABL>
9 002260                DIRDEF  <ENABL>
10 02270                DIRDEF  <END  >
11 02300                DIRDEF  <ENDC >,        DFLCND
12 02310                DIRDEF  <ENDM >,        DFLMAC, XMACRO
13 02320                DIRDEF  <ENDR >,        DFLMAC, XMACRO
14 02330                DIRDEF  <EOT  >
15 02340                DIRDEF  <ERROR>
16 02350                DIRDEF  <EVEN >
17 02360                DIRDEF  <FLT2 >,        DFLGEV, XFLTG
18 02370                DIRDEF  <FLT4 >,        DFLGEV, XFLTG
19 02400                DIRDEF  <GLOBL>
20 02410                DIRDEF  <IDENT>
21 02420                DIRDEF  <IF   >,        DFLCND
22 02430                DIRDEF  <IFDF >,        DFLCND
23 02440                DIRDEF  <IFEQ >,        DFLCND
24 02450                DIRDEF  <IFF  >,        DFLCND
25 02460                DIRDEF  <IFG  >,        DFLCND
26 02470                DIRDEF  <IFGE >,        DFLCND
27 02500                DIRDEF  <IFGT >,        DFLCND
28 02510                DIRDEF  <IFL  >,        DFLCND
29 02520                DIRDEF  <IFLE >,        DFLCND
30 02530                DIRDEF  <IFLT >,        DFLCND
31 02540                DIRDEF  <IFNDF>,        DFLCND
32 02550                DIRDEF  <IFNE >,        DFLCND
33 02560                DIRDEF  <IFNZ >,        DFLCND
34 02570                DIRDEF  <IFT  >,        DFLCND
35 02600                DIRDEF  <IFTF >,        DFLCND
36 02610                DIRDEF  <IFZ  >,        DFLCND
37 02620                DIRDEF  <IIF  >
38 02630                DIRDEF  <IRP  >,        DFLMAC, XMACRO
39 02640                DIRDEF  <IRPC >,        DFLMAC, XMACRO
40 02650                DIRDEF  <LIMIT>,        DFLGEV
41 02660                DIRDEF  <LIST >
```

```
PST     PERMANENT SYMBOL TABLE  MACRO V004A  PAGE 6


1 002670                DIRDEF  <MACR >,        DFLMAC, XMACRO
2 002700                DIRDEF  <MACRO>,        DFLMAC, XMACRO
3 002710                DIRDEF  <MCALL>,        DFLSMC, XMACRO
4 002720                DIRDEF  <MEXIT> ,       ,       XMACRO
5 002730                DIRDEF  <NARG > ,       ,       XMACRO
6 002740                DIRDEF  <NCHR > ,       ,       XMACRO
7 002750                DIRDEF  <NLIST>
8 002760                DIRDEF  <NTYPE> ,       ,       XMACRO
9 002770                DIRDEF  <ODD  >
10 03000                DIRDEF  <PAGE >
11 03010                DIRDEF  <PRINT>
12 03020                DIRDEF  <RADIX>
13 03030                DIRDEF  <RAD50>,        DFLGEV
14 03040                DIRDEF  <REM  >
15 03050                DIRDEF  <REPT >,        DFLMAC, XMACRO
16 03060                DIRDEF  <SBTTL>
17 03070                DIRDEF  <TITLE>
18 03100        WRDSYM:
19 03100                DIRDEF  <WORD >,        DFLGEV
20
21
22 03110        PSTTOP:                         ;TOP LIMIT
23
24      000001          .END


PST     PERMANENT SYMBOL TABLE  MACRO V004A  PAGE 6+
SYMBOL TABLE

ASCII = ****** G        ASCIZ = ****** G        ASECT = ****** G
BLKB  = ****** G        BLKW  = ****** G        BYTE  = ****** G
CSECT = ****** G        DFLCND= 000004 G        DFLGBM= 000010 G
DFLGEV= 000020 G        DFLMAC= 000002 G        DFLSMC= 000001 G
DR1   = 000200          DR2   = 000100          DSABL = ****** G
ENABL = ****** G        END   = ****** G        ENDC  = ****** G
ENDM  = ****** G        ENDR  = ****** G        EOT   = ****** G
ERROR = ****** G        EVEN  = ****** G        FLT2  = ****** G
FLT4  = ****** G        GLOBL = ****** G        IDENT = ****** G
IF    = ****** G        IFDF  = ****** G        IFEQ  = ****** G
IFF   = ****** G        IFG   = ****** G        IFGE  = ****** G
IFGT  = ****** G        IFL   = ****** G        IFLE  = ****** G
IFLT  = ****** G        IFNDF = ****** G        IFNE  = ****** G
IFNZ  = ****** G        IFT   = ****** G        IFTF  = ****** G
IFZ   = ****** G        IIF   = ****** G        IRP   = ****** G
IRPC  = ****** G        LIMIT = ****** G        LIST  = ****** G
MACR  = ****** G        MACRO = ****** G        MCALL = ****** G
MEXIT = ****** G        NARG  = ****** G        NCHR  = ****** G
NLIST = ****** G        NTYPE = ****** G        ODD   = ****** G
OPCL00= ****** G        OPCL01= ****** G        OPCL02= ****** G
OPCL03= ****** G        OPCL04= ****** G        OPCL05= ****** G
OPCL06= ****** G        OPCL07= ****** G        OPCL08= ****** G
OPCL09= ****** G        OPCL10= ****** G        OPCL11= ****** G
OPCL12= ****** G        OPCL13= ****** G        OPCL14= ****** G
PAGE  = ****** G        PRINT = ****** G        PSTBAS  000000RG    002
PSTTOP  003110RG    002 RADIX = ****** G        RAD50 = ****** G
REM   = ****** G        REPT  = ****** G        SBTTL = ****** G
TITLE = ****** G        WORD  = ****** G        WRDSYM  003100RG    002

. ABS.  000000      000
        000000      001
PSTSEC  003110      002

ERRORS DETECTED:  0
FREE CORE:  16991. WORDS
,LP:/CRF<PST,04A
```

APPENDIX D

LISTING OF SYSMAC.SML

(SYSTEM MACRO FILE)

```
;       PDP-11 DOS SYSTEM MACROS   V003A
;COPYRIGHT 1972 DIGITAL EQUIPMENT CORPORATION
;
;               JUNE 1, 1972.


        .MACRO  .PARAM
R0=%A00
R1=%A01
R2=%A02
R3=%A03
R4=%A04
R5=%A05
R6=%A06
R7=%A07
SP=%A06
PC=%A07
PSW=A0177776
SWR=A0177570
        .ENDM
        .MACRO  .INIT   .LBLCK
        .MCALL  .AMODE
        .AMODE  .LBLCK
        EMT <A06>
        .ENDM

        .MACRO  .RLSE   .LBLCK
        .MCALL  .AMODE
        .AMODE  .LBLCK
        EMT <A07>
        .ENDM

        .MACRO  .CLOSE  .LBLCK
        .MCALL  .AMODE
        .AMODE  .LBLCK
        EMT <A017>
        .ENDM

        .MACRO  .READ   .LBLCK,.LBUFF
        .MCALL  .AMODE
        .AMODE  .LBUFF
        .AMODE  .LBLCK
        EMT <A04>
        .ENDM
```

```
.MACRO   .WRITE   .LBLCK,.LBUFF
.MCALL   .AMODE
.AMODE   .LBUFF
.AMODE   .LBLCK
EMT <^O02>
.ENDM

.MACRO   .OPEND   .LBLCK,.FBLCK
.MCALL   .CODE,.OPEN
.CODE    .FBLCK,<^O02>
.OPEN    .LBLCK,.FBLCK
.ENDM

.MACRO   .OPENI   .LBLCK,.FBLCK
.MCALL   .CODE,.OPEN
.CODE    .FBLCK,<^O04>
.OPEN    .LBLCK,.FBLCK
.ENDM

.MACRO   .OPENU   .LBLCK,.FBLCK
.MCALL   .CODE,.OPEN
.CODE    .FBLCK,<^O01>
.OPEN    .LBLCK,.FBLCK
.ENDM

.MACRO   .OPENC   .LBLCK,.FBLCK
.MCALL   .CODE,.OPEN
.CODE    .FBLCK,<^O13>
.OPEN    .LBLCK,.FBLCK
.ENDM

.MACRO   .OPENE   .LBLCK,.FBLCK
.MCALL   .CODE,.OPEN
.CODE    .FBLCK,<^O03>
.OPEN    .LBLCK,.FBLCK
.ENDM

.MACRO   .OPEN    .LBLCK,.FBLCK
.MCALL   .AMODE
.AMODE   .FBLCK
.AMODE   .LBLCK
EMT <^O16>
.ENDM

.MACRO   .WAIT    .LBLCK
.MCALL   .AMODE
.AMODE   .LBLCK
EMT <^O1>
.ENDM

.MACRO   .WAITR   .LBLCK,.ADDR
.MCALL   .AMODE
.AMODE   .ADDR
.AMODE   .LBLCK
EMT <^O00>
.ENDM

.MACRO   .BLOCK   .LBLCK,.BBLCK
.MCALL   .AMODE
.AMODE   .BBLCK
.AMODE   .LBLCK
EMT <^O11>
.ENDM
```

```
.MACRO  .TRAN   .LBLCK,.TBLCK
.MCALL  .AMODE
.AMODE  .TBLCK
.AMODE  .LBLCK
EMT <^O10>
.ENDM

.MACRO  .SPEC   .LBLCK,.SARG
.MCALL  .AMODE
.AMODE  .SARG
.AMODE  .LBLCK
EMT <^O12>
.ENDM

.MACRO  .STAT   .LBLCK
.MCALL  .AMODE
.AMODE  .LBLCK
EMT <^O13>
.ENDM

.MACRO  .ALLOC  .LBLCK,.FBLCK,.N
.MCALL  .AMODE
.AMODE  .N
.AMODE  .FBLCK
.AMODE  .LBLCK
EMT <^O15>
.ENDM

.MACRO  .DELET  .LBLCK,.FBLCK
.MCALL  .AMODE
.AMODE  .FBLCK
.AMODE  .LBLCK
EMT <^O21>
.ENDM

.MACRO  .RENAM  .LBLCK,.OFB,.NFB
.MCALL  .AMODE
.AMODE  .NFB
.AMODE  .OFB
.AMODE  .LBLCK
EMT <^O20>
.ENDM

.MACRO  .APPEND .LBLCK,.1FB,.2FB
.MCALL  .AMODE
.AMODE  .2FB
.AMODE  .1FB
.AMODE  .LBLCK
EMT <^O2>
.ENDM

.MACRO  .LOOK   .LBLCK,.FBLCK,.OP
.MCALL  .AMODE
.AMODE  .FBLCK
.IIF NB,.OP,CLR -(SP)
.AMODE  .LBLCK
EMT <^O14>
.ENDM

.MACRO  .KEEP   .LBLCK,.FBLCK
.MCALL  .AMODE
.AMODE  .FBLCK
.AMODE  .LBLCK
EMT <^O24>
.ENDM
```

```
.MACRO   .EXIT
EMT <^O60>
.ENDM

.MACRO   .TRAP    .STUS,.ADDR
.MCALL   .AMODE
.AMODE   .ADDR
.AMODE   .STUS
MOV      #^O1,-(SP)
EMT <^O41>
.ENDM

.MACRO   .STFPU   .STUS,.ADDR
.MCALL   .AMODE
.AMODE   .ADDR
.AMODE   .STUS
MOV      #^O3,-(SP)
EMT <^O41>
.ENDM

.MACRO   .RECRD   .LBLCK,.RBLCK
.MCALL   .AMODE
.AMODE   .RBLCK
.AMODE   .LBLCK
EMT <^O25>
.ENDM

.MACRO   .DUMP    .LOW,.HIGH,.CDE
.MCALL   .AMODE
.AMODE   .LOW
.AMODE   .HIGH
.AMODE   .CDE
EMT <^O64>
.ENDM

.MACRO   .RSTRT   .ADDR
.MCALL   .AMODE
.AMODE   .ADDR
MOV      #^O2,-(SP)
EMT <^O41>
.ENDM

.MACRO   .CORE
MOV      #^O100,-(SP)
EMT <^O41>
.ENDM

.MACRO   .MONR
MOV      #^O101,-(SP)
EMT <^O41>
.ENDM

.MACRO   .MONF
MOV      #^O102,-(SP)
EMT <^O41>
.ENDM

.MACRO   .DATE
MOV      #^O103,-(SP)
EMT <^O41>
.ENDM
```

```
.MACRO  .TIME
MOV     #^O104,-(SP)
EMT <^O41>
.ENDM

.MACRO  .GTUIC
MOV     #^O105,-(SP)
EMT <^O41>
.ENDM

.MACRO  .SYSDV
MOV     #^O106,-(SP)
EMT <^O41>
.ENDM

.MACRO  .RADPK  .ADDR
.MCALL  .AMODE
.AMODE  .ADDR
CLR     -(SP)
EMT <^O42>
.ENDM

.MACRO  .RADUP  .ADDR,.WRD
.MCALL  .AMODE
.AMODE  .WRD
.AMODE  .ADDR
MOV     #^O1,-(SP)
EMT <^O42>
.ENDM

.MACRO  .D2BIN  .ADDR
.MCALL  .AMODE
.AMODE  .ADDR
MOV     #^O2,-(SP)
EMT <^O42>
.ENDM

.MACRO  .BIN2D  .ADDR,.WRD
.MCALL  .AMODE
.AMODE  .WRD
.AMODE  .ADDR
MOV     #^O3,-(SP)
EMT <^O42>
.ENDM

.MACRO  .O2BIN  .ADDR
.MCALL  .AMODE
.AMODE  .ADDR
MOV     #^O4,-(SP)
EMT <^O42>
.ENDM

.MACRO  .BIN2O  .ADDR,.WRD
.MCALL  .AMODE
.AMODE  .WRD
.AMODE  .ADDR
MOV     #^O5,-(SP)
EMT <^O42>
.ENDM
```

```
.MACRO  .CSI1   .CMDBF
.MCALL  .AMODE
.AMODE  .CMDBF
EMT <^O56>
.ENDM

.MACRO  .CSI2   .CSBLK
.MCALL  .AMODE
.AMODE  .CSBLK
EMT <^O57>
.ENDM

.MACRO  .DTCVT  .ADDR
.MCALL  .CVTDT
.CVTDT  #^O0,.ADDR
.ENDM

.MACRO  .TMCVT  .ADDR
.MCALL  .CVTDT
.CVTDT  #^O1,.ADDR
.ENDM

.MACRO  .CVTDT  .CDE,.ADDR,.VAL1,.VAL2
.MCALL  .AMODE
.IF     NB,.VAL2
.AMODE  .VAL2
.ENDC
.IF     NB,.VAL1
.AMODE  .VAL1
.ENDC
.AMODE  .ADDR
.AMODE  .CDE
EMT <^O66>
.ENDM

.MACRO  .GTPLA
CLR     -(SP)
MOV     #^O5,-(SP)
EMT <^O41>
.ENDM

.MACRO  .STPLA  .ADDR
.MCALL  .AMODE
.AMODE  .ADDR
MOV     #^O5,-(SP)
EMT <^O41>
.ENDM

.MACRO  .GTCIL
MOV     #^O107,-(SP)
EMT <^O41>
.ENDM

.MACRO  .GTSTK
CLR     -(SP)
MOV     #^O4,-(SP)
EMT <^O41>
.ENDM

.MACRO  .STSTK  .ADDR
.MCALL  .AMODE
.AMODE  .ADDR
```

```
        MOV     #^O04,-(SP)
        EMT <^O41>
        .ENDM

        .MACRO  .RUN    .RNBLK
        .MCALL  .AMODE
        .AMODE  .RNBLK
        EMT <^O65>
        .ENDM

        .MACRO  .FLUSH  .CDE
        .MCALL  .AMODE
        .AMODE  .CDE
        EMT <^O67>
        .ENDM

; THE MACRO .AMODE ACCEPTS ONE ARGUMENT AND
; AS A FUNCTION OF THE ADDRESSING MODE OF
; THE ARGUMENT GENERATES THE APPROPRIATE
; MOV TO -(SP).
; ADDRESS MODES THAT ARE TROUBLESOME (E.G.
; X(SP)) OR UNLIKELY (E.G. SP) WILL RESULT
; IN A .ERROR TO CMO INCLUDING THE
; VALUE OF THE ADDRESS MODE (E.G. X(SP)
; IS REPRESENTED AS 000066), THE ARGUMENT ITSELF
; AND THE TEXT "ADDRESSING MODE ILLEGAL AS SYSTEM
; MACRO ARGUMENT".
;
        .MACRO  .AMODE  .ARG
SP=%^O06
        .NTYPE  .SYM,.ARG       ;.SYM=ADDRESS MODE.

        .IF LE,.SYM-^O05
        MOV     .ARG,-(SP)      ;R0 TO R5
        .MEXIT
        .ENDC

        .IF EQ,.SYM&^O70-^O10
        .IF LE,.SYM&^O07-^O06
        MOV     .ARG,-(SP)      ;@R0 TO @R6
        .MEXIT
        .ENDC
        .ENDC

        .IF EQ,.SYM&^O60-^O20
        MOV     .ARG,-(SP)      ;[@](R0)+ TO [@](R7)+
        .MEXIT                  ; #N,@#ADDR
        .ENDC

        .IF EQ,.SYM&^O40-^O40
        .IF LE,.SYM&^O07-^O05
        MOV     .ARG,-(SP)      ;[@]-(R0) TO [@]-(R5)
        .MEXIT                  ;[@]X(R0) TO [@]X(R5)
        .ENDC
        .ENDC

        .IF EQ,.SYM&^O67-^O67
        MOV     .ARG,-(SP)      ;ADDR AND @ADDR
        .MEXIT
        .ENDC
```

```
        .ERROR .SYM             ;.ARG ADDRESSING MODE ILLEGAL
        .PRINT                  ;AS SYSTEM MACRO ARGUMENT.
        .ENDM

; THE MACRO .CODE SETS UP THE FILEBLOCK
; WITH THE HOW OPEN CODE.
; THE ADDRESS OF THE FILEBLOCK MUST
; BE IN A REGISTER (R0 TO R5)

        .MACRO  .CODE   .FBLK,.N
        .NTYPE  .SYM,.FBLK

        .IF LE,.SYM-A05
        MOVB    #.N,-A02(.FBLK) ;R0 TO R5
        .MEXIT
        .ENDC

        .ERROR .SYM             ;.FBLK ADDRESSING MODE ILLEGAL
        .PRINT                  ;FOR .OPEN FILE BLOCK
        .ENDM
```

APPENDIX E

ERROR MESSAGE SUMMARY

E.1 MACRO-11 ERROR CODES

MACRO-11 error codes are printed following a field of six asterisk characters and on the line preceeding the source line containing the error. For example:

```
******A
26 ØØ236  ØØØØØ2'    .WORD REL1+REL2
```

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

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

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

E.2 SYSTEM ERROR MESSAGES

Error Code	Meaning
S217	Insufficient core space.
S202	Binary or listing device full.
S203	Illegal switch Too many switches Illegal switch value Too many switch values
S204	Too many output file specifications
S206	No source files specified.

INDEX

HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections are published by Software Information Service in the following newsletters.

Digital Software News for the PDP-8 & PDP-12
Digital Software News for the PDP-11
Digital Software News for the PDP-9/15 Family

These newsletters contain information applicable to software available from Digital's Program Library, Articles in Digital Software News update the cumulative Software Performance Summary which is contained in each basic kit of system software for new computers. To assure that the monthly Digital Software News is sent to the appropriate software contact at your installation, please check with the Software Specialist or Sales Engineer at your nearest Digital office.

Questions or problems concerning Digital's Software should be reported to the Software Specialist. In cases where no Software Specialist is available, please send a Software Performance Report form with details of the problem to:

Software Information Service
Digital Equipment Corporation
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

These forms which are provided in the software kit should be fully filled out and accompanied by teletype output as well as listings or tapes of the user program to facilitate a complete investigation. An answer will be sent to the individual and appropriate topics of general interest will be printed in the newsletter.

Orders for new and revised software and manuals, additional Software Performance Report forms, and software price lists should be directed to the nearest Digital Field office or representative. U.S.A. customers may order directly from the Program Library in Maynard. When ordering, include the code number and a brief description of the software requested.

Digital Equipment Computer Users Society (DECUS) maintains a user library and publishes a catalog of programs as well as the DECUSCOPE magazine for its members and non-members who request it. For further information please write to:

DECUS
Digital Equipment Corporation
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

— — — — — — — — — — — — — — — — — Fold Here — — — — — — — — — — — — — — — — — —

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

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

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

Postage will be paid by:

digital

Digital Equipment Corporation
Software Information Services
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754