PDP-11
Paper Tape Software Handbook

Order No. DEC-11-XPTSA-B-D

digital equipment corporation · maynard, massachusetts

First Printing, April 1970
Revised: March 1971
January 1972
February 1973
June 1975
April 1976

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

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

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL
DEC
PDP
DECUS
UNIBUS
COMPUTER LABS
COMTEX
DDT
DECCOMM

DECsystem-10
DECtape
DIBOL
EDUSYSTEM
FLIP CHIP
FOCAL
INDAC
LAB-8
DECsystem-20

MASSBUS
OMNIBUS
OS/8
PHA
RSTS
RSX
TYPESET-8
TYPESET-10
TYPESET-11

5/76-34

CONTENTS

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

Page

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

Page

FIGURES

TABLES

CHAPTER 1

PAL-11S ASSEMBLY LANGUAGE AND ASSEMBLER

PAL-11S Assembly (Program Assembly Language for the PDP-11, Relocatable, Stand Alone Version) enables you to write source (symbolic) programs using letters, numbers, and symbols which are meaningful to you. The source programs, generated either on-line using the Text Editor (ED-11), or off-line, are then assembled into object modules which are processed by the PDP-11 linker, LINK-11S. LINK-11S produces a load module which is loaded by the Absolute Loader for execution. Object modules may contain absolute and/or relocatable code and separately assembled object modules may be linked with global symbols. The object module is produced after two passes through the Assembler; an optional third pass produces a complete octal/symbolic listing of the assembled program. This listing is especially useful for documentation and debugging purposes.

This chapter not only explains how to write PAL-11S programs but also how to assemble the source programs into object modules. All facets of the assembly language are explained and illustrated with many examples, and the chapter concludes with assembling procedures. In explaining how to write PAL-11S source programs, it is necessary, especially at the outset, to make frequent forward references. Therefore, we recommend that you first read through the entire chapter to get a "feel" for the language, and then reread the chapter, this time referring to appropriate sections as indicated, for a thorough understanding of the language and assembling procedures.

Some notable features of PAL-11S are:

1. Selective assembly pass functions.

2. Device specification for pass functions.

3. Optional error listing on the teleprinter.

4. Double buffered and concurrent I/O (provided by IOXLPT).

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.

The PAL-11S Assembler requires 8K of memory and provides for about 900 user-defined symbols (see Section 1.3.2). In addition, it allows a line printer to be used for program listing and/or symbol table listing.

PAL-11S ASSEMBLY LANGUAGE AND ASSEMBLER

The following discussion of the PAL-11S Assembly Language assumes that you have read the PDP-11 Processor Handbook with emphasis on those sections which deal with the PDP-11 instruction repertoire, formats, and timings -- a thorough knowledge of these is vital to efficient assembly language programming.

1.1 CHARACTER SET

A PAL-11S source program is composed of symbols, numbers, expressions, symbolic instructions, assembler directives, argument separators, and line terminators written using the following ASCII[1] characters.

1. The letters A through Z. (Upper and lower case letters are acceptable, although upon input, lower case letters will be converted to upper case letters.)

2. The numbers 0 through 9.

3. The characters . and $. (These characters are reserved for systems use.)

4. The separating or terminating symbols:

   : = % # @ ( ) , ; " ' + - & !

   carriage return tab space line feed form feed

1.2 STATEMENTS

A source program is composed of a sequence of statements, where each statement is on a single line. The statement is terminated by a carriage return character which must be immediately followed by either a line feed or form feed character. Should a carriage return character be present and not be followed by a line feed or form feed, the Assembler will generate a Q error (Section 1.10), and that portion of the line following the carriage return will be ignored. Since the carriage return terminator is a required statement terminator, a line feed or form feed not immediately preceded by a carriage return will have one inserted by the Assembler.

It should be noted that, if the Editor (ED-11) is being used to create the source program, a typed carriage return (RETURN key) automatically generates a line feed character.

A statement may be composed of up to four fields which are identified by their order of appearance and by specified terminating characters as explained below and summarized in Appendix B. The four fields are:

Label Operator Operand Comment

The label and comment fields are optional. The operator and operand fields are inter-dependent -- either may be omitted depending upon the contents of the other.

[1] ASCII stands for American Standard Code for Information Interchange.

1.2.1 Label

A label is a user-defined symbol (see Section 1.3.2) which is assigned the value of the current location counter. This value may be either absolute or relocatable depending on whether the location counter value is absolute or relocatable. In the latter case, the final absolute value is assigned by the Linker, i.e., the value + 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 the statement:

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

will assign the value 100 to the label ABCD so that subsequent reference to ABCD will be to location 100 . In the above case if the location counter were relocatable then the final value of ABCD would be 100 +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 will have the same value. For example, if the current location counter is 100 , multiple labels in the statement:

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

will equate each of the three labels ABC, $DD, and A7.7 with the value 100 ($ and . are reserved for system software).

The error code M (multiple definition of a symbol) will be generated during assembly if two or more labels have the same first six characters.

1.2.2 Operator

An operator follows the label field in a statement, and may be an instruction mnemonic or an assembler directive (see Section 1.8 and Appendix B). When it is an instruction mnemonic, it specifies what action is to be performed on any operand(s) which follows it. When it is an assembler directive, it specifies a certain function or action to be performed during assembly.

The operator may be preceded only by one or more labels and may be followed by one or more operands and/or a comment. An operator is legally terminated by a space, tab, or any of the following characters:

```
# + - @ ( " ' % ! & , ;

line feed     form feed     carriage return
```

The use of each character above will be explained in this chapter.

Consider the following examples:

```
MOV→|A,B      ;→|(TAB) terminates operator MOV
MOV@A,B       ;@ terminates operator MOV
```

When the operator stands alone without an operand or comment, it is terminated by a carriage return followed by a line feed or form feed character.

1.2.3 Operand

An operand is that part of a statement which is operated on by the operator -- an instruction mnemonic or assembler directive. Operands may be symbols, expressions, or numbers. When multiple operands appear within a statement, each is separated from the next by a comma. An operand may be preceded by an operator and/or label, and followed by a comment.

The operand field is terminated by a semicolon when followed by a comment, or by a carriage return followed by a line feed or form feed character when the operand ends the statement. For example,

```
          LABEL:         MOV GEORGE,BOB        ;THIS IS A COMMENT
```

where the space between MOV and GEORGE terminated the operator field and began the operand field; the comma separated the operands GEORGE and BOB; the semicolon terminated the operand field and began the comment.

1.2.4 Comments

The comment field is optional and may contain any ASCII character except null, rubout, carriage return, line feed or form feed. All other characters, even those with special significance are ignored by the Assembler when used in the comment field.

The comment field may be preceded by none, any, or all of the other three fields. It must begin with the semicolon and end with a carriage return followed by a line feed or form feed character. For example,

```
          LABEL:         CLR HERE          ;THIS IS A $1.00 COMMENT
```

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

1.2.5 Format Control

The format is controlled by the space and tab characters. They have no effect on the assembling process of the source program unless they are embedded within a symbol, number, or ASCII text; or are used as the operator field terminator. Thus, they can be used to provide a neat, readable 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 much easier to read.

Page size is controlled by the form feed character. A page of n lines is created by inserting a form feed (CTRL/FORM keys on the keyboard) after the nth line. If no form feed is present, a page is automatically terminated after 56 lines.

1.3 SYMBOLS

There are two types of symbols, permanent and user-defined. Both are stored in the Assembler's symbol table. Initially, the symbol table contains the permanent symbols, but as the source program is assembled, user-defined symbols are added to the table.

1.3.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics (see Appendix B.3) and assembler directives (see Section 1.8). These symbols are a permanent part of the Assembler's symbol table and need not be defined before being used in the source program.

1.3.2 User-Defined Symbols

User-defined symbols are those defined as labels (see Section 1.2.1) or by direct assignment (see Section 1.3.3). These symbols are added to the symbol table as they are encountered during the first pass of the assembly. They can be composed of alphanumeric characters, dollar signs, and periods only; again $'s and .'s are reserved for system software. Any other character is illegal and, if used, will result in the error message I or QU (see Section 1.10). I is a low priority error which may be flagged as QU first. The following rules also apply to user-defined symbols:

1. The first character must not be a number.
2. Each symbol must be unique within the first six characters.
3. A symbol may be written with more than six legal characters but the seventh and subsequent characters are only checked for legality, and are not otherwise recognized by the Assembler.
4. Spaces and tabs must not be embedded within a symbol.

A user-defined symbol may duplicate a permanent symbol. The value associated with a permanent symbol that is also user-defined depends upon its use:

1. A permanent symbol encountered in the operator field is associated with its corresponding machine op-code.
2. If a permanent symbol in the operand field is also user-defined, its user-defined value is associated with the symbol. If the symbol is not found to be user-defined, then the corresponding machine op-code value is associated with the symbol.

User-defined symbols are either internal or global. All symbols are internal unless they are explicitly typed as global with the .GLOBL assembler directive (see Section 1.8.2). Global symbols are used to provide links between object modules. A global symbol which is defined (as a label or by direct assignment) in a program is called an entry symbol or entry point. Such symbols may be referred to from other object modules or assemblies. A global symbol which is not defined in the current assembly is called an external symbol. Some other assembly must define the same symbol as an entry point.

1.3.3 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 Assembler's 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 newly assigned value will replace the previous value assigned to the symbol.

The symbol takes on the relocatable or absolute attribute of the defining expression. However, if the defining expression is global, the defined symbol will not be global unless previously defined as such (see Section 1.4).

The general format for a direct assignment statement is:

```
symbol = expression.
```

The following conventions apply:

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

Example of two levels of forward referencing (illegal):

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

X and Y are both undefined throughout pass 1 and will be listed on the teleprinter as such at the end of that pass. X is undefined throughout pass 2, and will cause a U error message.

Examples:

```
        A=1                 ;THE SYMBOL A IS EQUATED WITH THE VALUE 1

        B='A-1&MASKLOW      ;THE SYMBOL B IS EQUATED WITH THE EXPRESSION'S
                            ;VALUE

C:      D=3                 ;THE SYMBOL D IS EQUATED WITH 3.  THE
E:      MOV #1,ABLE         ;LABELS C AND E ARE EQUATED WITH THE
                            ;NUMERICAL MEMORY ADDRESS OF THE MOV
                            ;COMMAND
```

1.3.4 Register Symbols

The eight general registers of the PDP-11 are numbered 0 through 7. These registers may be referenced by use of a register symbol; that is, a symbolic name for a register. A register symbol is defined by means of a direct assignment, where the defining expression contains at least one term preceded by a % or at least one term previously defined as a register symbol. In addition, the defining expression of a register symbol must be absolute. For example:

```
        R0=%0           ;DEFINE R0 AS REGISTER 0

        R3=R0+3         ;DEFINED R3 AS REGISTER 3

        R4=1+%3         ;DEFINE R4 AS REGISTER 4

        THERE=%2        ;DEFINE 'THERE' AS REGISTER 2
```

It is important to note that all register symbols must be defined before they are referenced. A forward reference to a register symbol will generally cause phase errors (see Section 1.10).

The % may be used in any expression thereby indicating a reference to a register. Such an expression is a register expression. Thus, the statement:

```
          CLR     %6
```

will clear register 6 while the statement:

```
          CLR     6
```

will clear the word at memory address 6. 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 and are thoroughly explained in Sections 1.7.11 and 1.7.12. Two obvious examples of this are:

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

        CLR    X(2)       ;ANY EXPRESSION ENCLOSED IN ( ) MUST BE
                          ;A REGISTER.  IN THIS CASE, INDEX REGISTER
                          ;2
```

1.4 EXPRESSIONS

Arithmetic and logical operators (see Section 1.4.2) may be used to form expressions. A term of an expression may be a permanent or user-defined symbol (which may be absolute, relocatable or global), a number, ASCII data, or the present value of the assembly location counter represented by the period (see Section 1.5). Expressions are evaluated from left to right. Parenthetical grouping is not allowed.

Expressions are evaluated as word quantities. The operands of a .BYTE directive (Section 1.8.8) are evaluated as word expressions before truncation to the low-order eight bits. The evaluation of an expression includes the evaluation of the mode of the resultant expression; that is, absolute, relocatable or external. The definition of the modes of expression are given below in Section 1.4.4.

A missing term, expression or external symbol will be interpreted as 0. A missing operator will be interpreted as +. The error code Q (Questionable syntax) will be generated for a missing operator. For example,

```
          A + -100             ;OPERAND MISSING
```

will be evaluated as A + 0 - 100, and

```
          TAG ! LA 177777      ;OPERATOR MISSING
```

will be evaluated as TAG ! LA+177777.

The value of an external expression will be the value of the absolute part of the expression; e.g., EXT+A will have a value of A. This will be modified by the linker to become EXT+A.

1.4.1 Numbers

The Assembler accepts both octal and decimal numbers. Octal numbers consist of the digits 0 through 7 only. Decimal numbers consist of the digits 0 through 9 followed by a decimal point. If a number contains an 8 or 9 and is not followed by a decimal point, the N error code (see Section 1.10) will be printed and the number will be interpreted as decimal. Negative numbers may be expressed as a number preceded by a minus sign rather than in a two's complement form. Positive numbers may be preceded by a plus sign although this is not required.

If a number is too large to fit into 16 bits, the number is truncated from the left. In the assembly listing the statement will be flagged with a Truncation (T) error. Numbers are always considered to be absolute quantities (that is, not relocatable).

1.4.2 Arithmetic And Logical Operators

The arithmetic operators are:

+	indicates addition or a positive number
-	indicates subtraction or a negative number

The logical operators are:

&	indicates the logical AND operation
!	indicates the logical inclusive OR operation

AND	OR
0 & 0 = 0	0 ! 0 = 0
0 & 1 = 0	0 ! 1 = 1
1 & 0 = 0	1 ! 0 = 1
1 & 1 = 1	1 ! 1 = 1

1.4.3 ASCII Conversion

When preceded by an apostrophe, any ASCII character (except null, rubout, carriage return, line feed, or form feed) is assigned the 7-bit ASCII value of the character (see Appendix A). For example,

```
'A
```

is assigned the value 101_8.

When preceded by a quotation mark, two ASCII characters (not including null, rubout, carriage return, line feed, or form feed) are assigned the 7-bit ASCII values of each of the characters to be used. Each 7-bit value is stored in an 8-bit byte and the bytes are combined to form a word. For example "AB will store the ASCII value of A in the low-order (even) byte and the value of B in the high-order (odd) byte:

```
              high-order byte   |      low-order byte
B's value =      1    0    2    |       1    0    1      = A's value

              0  100  001      001          000  001

              0   4    1        1            0    1
                   "AB=041101   |
```

ASCII text is always absolute.

1.4.4 Mode of Expressions

The mode of an expression may be absolute, relocatable or external as defined below:

A term of an expression is absolute, relocatable or external depending on whether its definer (i.e., number, symbol, etc.) is absolute, relocatable or external. Numbers, permanent symbols and generated data are always treated as absolute.

An absolute expression is defined as:

1. Absolute term (one whose value is defined at assembly time) preceded optionally by a single plus or minus sign, or

2. Relocatable expression minus a relocatable term, or

3. Absolute expression followed by an operator followed by an absolute expression.

A relocatable expression is defined as:

1. Relocatable term (one whose value is not known until link time), or

2. Relocatable expression followed by an arithmetic operator followed by an absolute expression, or

3. Absolute expression followed by a plus operator followed by a relocatable expression.

An external expression is defined as:

1. External term (one whose value is defined outside the program), or

2. External expression followed by an arithmetic operator followed by an absolute term, or

3. Absolute expression followed by a plus operator followed by an external expression.

In the following examples:

ABS is an absolute symbol,

REL is a relocatable symbol,

EXT is an external symbol.

Examples:

The following are valid expressions:

```
        EXT + ABS             ;External expression

        REL+REL-REL           ;Relocatable expression

        ABS+REL-REL & ABS     ;Absolute expression
```

The following are illegal expressions:

```
        EXT+REL
        REL+REL
        ABS-EXT
```

1.5 ASSEMBLY LOCATION COUNTER

The period (.) is the symbol for the assembly location counter. (Note difference of Program Counter. ≠ PC. See Section 1.7.) When used in the operand field of an instruction, it represents the address of the first word of the instruction. When used in the operand field of an assembler directive, it represents the address of the current byte or word. For example,

```
        A:  MOV #.,R0         ;.refers to location A,
                              ;i.e., the address of the
                              ;MOV instruction
```

(# is explained in Section 1.7.9.)

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. However, the mode cannot be external. Neither can one change the existing mode of the location counter 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 1.8.3.

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

Examples:

```
        .ASECT
                .=500           ;SET LOCATION COUNTER TO ABSOLUTE 500
   FIRST:   MOV   .+10,COUNT ;THE LABEL FIRST HAS THE VALUE 500
                              ;(OCTAL) .+10 EQUALS 510 (OCTAL).  THE
                              ;CONTENTS OF LOCATION 510 (OCTAL) WILL
                              ;BE DEPOSITED IN LOCATION COUNT.

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

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

.CSECT

         .=.+20             ;SET LOCATION COUNTER TO RELOCATABLE
                            ;20.

THIRD:   .WORD 0            ;THE LABEL THIRD HAS THE VALUE OF
                            ;RELOCATABLE 20.
```

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

```
        .=.+100
```

will reserve 100 bytes of storage space in the program. The next instruction will be stored at 1100.

1.6 RELOCATION AND LINKING

The output of the relocatable assembler is an object module which must be processed by the PDP-11 Linker, LINK-11S, before loading and execution. The Linker essentially fixes (i.e., makes absolute) the values of external or relocatable symbols and creates another module (load module) which contains the binary data to be loaded and executed.

To enable the Linker to fix the value of an expression the assembler issues certain directives to the Linker together with the required parameters. In the case of relocatable expressions the Linker adds the base of the relocatable section (the location in memory of relocatable 0) 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 being linked and adds it to the value of the external expression provided by the Assembler.

All instructions that are to be modified as described above will be marked by a single apostrophe in the assembly listing. Thus the binary text output will look as follows for the given examples:

```
005065'  CLR   EXTERNAL(5)        ;
000000                            ;VALUE OF EXTERNAL SYMBOL
                                  ;ASSUMED ZERO; WILL BE
                                  ;MODIFIED BY THE LINKER.
005065'  CLR   EXTERNAL+6(5)      ;
000006                            ;

005065'  CLR   RELOCATABLE(5)     ;ASSUMING WE ARE IN THE
000040                            ;ABSOLUTE SECTION AND
                                  ;THE VALUE OF RELOCATABLE
                                  ;IS RELOCATABLE 40
```

1.7 ADDRESSING

The Program Counter (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 (see sections 1.7.7, 1.7.8 and 1.7.10), the processor uses the Program Counter to fetch the base from memory. Hence, using the rule above, the PC increments by two, and now points to the next word.

The following conventions are used in this section:

1. Let E be any expression as defined in Section 1.4.

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

Examples:

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

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

4. Let A be a general address specification which produces a 6-bit mode address field as described in a PDP-11 Processor Handbook.

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

1.7.1 Register Mode

The register contains the operand.

Format: R

Example:

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

1.7.2 Deferred Register Mode

The register contains the address of the operand.

Format: @R or (ER)

Example:

```
CLR@R1     ;CLEAR THE WORD AT THE
 or        ;ADDRESS CONTAINED IN
CLR(1)     ;REGISTER 1
```

1.7.3 Autoincrement Mode

The contents of the register are incremented immediately after being used as the address of the operand.

Format: (ER)+

Examples:

```
CLR    (R0)+    ;CLEAR WORDS AT ADDRESSES
CLR  (R0+3)+    ;CONTAINED IN REGISTERS 0,3, AND 2
CLR     (2)+    ;AND INCREMENT REGISTER CONTENTS
                ;BY TWO.
```

NOTE

Both JMP and JSR instructions using mode 2 (non-deferred autoincrement mode), execute differently on different PDP-11 processors. Avoid use of these instructions with mode 2 addressing.

Double operand instructions of the addressing form %R, (R)+ or %R, -(R) where the source and destination registers are the same, give different results on different PDP-11 processors, and should be avoided.

1.7.4 Deferred Autoincrement Mode

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

Format: @(ER)+

Example:

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

1.7.5 Autodecrement Mode

The contents of the register are decremented before being used as the address of the operand (see note in Section 1.7.3).

Format: -(ER)

Examples:

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

1.7.6 Deferred Autodecrement Mode

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

Format: @-(ER)

Example:

```
CLR @-(2)            ;DECREMENT CONTENTS OF REG. 2
                     ;BEFORE USING AS POINTER TO ADDRESS
                     ;OF WORD TO BE CLEARED.
```

1.7.7 Index Mode

Format: E(ER)

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.

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
```

1.7.8 Deferred Index Mode

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

Format: @E(ER)

Example:

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

1.7.9 Immediate Mode and Deferred Immediate (Absolute) 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: #E

Examples:

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

The operation of this mode is explained as follows:

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

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

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

If the #E is preceded by @, E specifies an absolute address.

1.7.10 Relative and Deferred Relative Modes

Relative mode is the normal mode for memory references.

Format: E

Examples:

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

This mode is assembled as Index mode, using 7, the PC, as the 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. Rather, it is the number which, when added to the PC, becomes the address of the operand. Thus, the base is X-PC. The operation is explained as follows:

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

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

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

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

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

This mode is called relative because the operand address is calculated relative to the current PC. The base is the distance (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.

If E is preceded by @ the expression's value is the pointer to the address of the operand.

1.7.11 Table of Mode Forms and Codes (6-bit(A) format only - see Section 1.7.12)

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
None of these forms increases the instruction length.	R	0n	Register
	@R or (ER)	1n	Register deferred
	(ER)+	2n	Autoincrement
	@(ER)+	3n	Autoincrement deferred
	-(ER)	4n	Autodecrement
	@-(ER)	5n	Autodecrement deferred

	Form	Mode	Meaning
Any of these forms adds a word to the instruction length.	E(ER)	6n	Index
	@E(ER)	7n	Index deferred
	#E	27	Immediate
	@#E	37	Absolute memory reference
	E	67	Relative
	@E	77	Relative deferred reference

Notes:

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

2. The form @#E differs from the form E in that the second or third word of the instruction contains the absolute address of the operand rather than the relative distance between the operand and the PC. Thus, the statement CLR @#100 will clear location 100 even if the instruction is moved from the point at which it was assembled.

The Assembler is not particular about left and right and dangling + and - signs in address fields. The following are some examples of incorrect syntax that assemble as indicated, without an error indication.

Form	Assembles As:	Form	Assembles As:
(R2)A	A(R2)	(R2)-	-(R2)
A-(R2)	A(R2) or A-0(R2)	@ (R2)A	@ A(R2)
A(Rw)+	A(R2)	A(R2)+B	A+B(R2)
+(R2)	(R2)+		

1.7.12 Instruction Forms

The instruction mnemonics are given in Appendix B. This section defines the number and nature of the operand fields for these instructions.

In the table that follows, let R, E, and ER represent expressions as defined in Sections 1.4 and 1.7 and let A be a 6-bit address specification of the forms:

E	@E	-(ER)	@ -(ER)
R	@R or (R)	E(ER)	@ E(ER)
(ER)+	@(ER)+	#E	@ #E

Table 1-1
Instruction Operand Fields

Instruction	Form	Example
Double Operand	Op A,A	MOV (R6)+, @Y
Single Operand	Op A	CLR -(R2)
OPERATE	OP	HALT
Branch	Op E	BR X+2
		BLO .-4
	where -128<(E-.-2)/2<127	
Subroutine Call	JSR ER,A	JSR PC,SUBR
Subroutine Return	RTS ER	RTS PC
EMT/TRAP	Op or Op E	EMT
	where 0<E<377	EMT 31

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:

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

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

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

Since PC = .+2, we have

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

NOTE

> It is illegal to branch to a location specified as an external symbol, or to a relocatable symbol when within an absolute section, or to an absolute symbol when 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 Truncation (T) error occurs.

Do not try to micro-program the condition code operators (see Appendix B, B.4). This makes sense in the PDP-11 hardware; however, the current PAL-11S Assembler does not support this capability. Thus:

```
          CLC!CLV
```

results in a Q error (see Appendix B, B.5) and the statement is assembled as CLC.

Expressions in the Assembler do, however, allow logical operators and the use of instruction mnemonics. Thus, the proper ways to write the above statement:

```
          .WORD CLC!          ;Operand of .WORD
          +CLC!CLV            ;Operand of default .WORD
          !CLC!CLV            ;Operand of default .WORD
```

1.8 ASSEMBLER DIRECTIVES

Assembler directives (sometimes called pseudo-ops) direct the assembly process and may generate data.

Assembler directives may be preceded by a label and followed by a comment. The assembler directive occupies the operator field. Only one directive may be placed in any one statement. One or more operands may occupy the operand field or it may be void -- allowable operands vary from directive to directive.

1.8.1 .TITLE

The .TITLE directive is used to name the object module. The name is assigned by the first symbol following the directive. If there is no .TITLE statement the default name assigned is ".MAIN.".

1.8.2 .GLOBL

The .GLOBL directive is used to declare a symbol as being global. It may be an entry symbol, in which case it is defined in the program, or it may be a external symbol, in which case it should be defined in another program which will be linked with this program by the linker.

The form of the .GLOBL directive is

```
          .GLOBL   NAMA, NAMB,...,NAMN
```

NOTE

> A symbol cannot be declared global by defining it as a global expression in a direct assignment statement.

If an illegal character is detected in the operand field of a .GLOBL statement, an error message is not generated; and the Assembler may ignore the remainder of the statement. Thus:

```
GLOBL A,B,@C,D
```

assembles without error as:

```
.GLOBL A,B
```

1.8.3 Program Section Directives (.ASECT and .CSECT)

The relocatable assembler provides for two program sections, an absolute section declared by an .ASECT directive and a relocatable section declared by a .CSECT directive. These directives therefore enable the programmer to specify that parts of his program be assembled in the absolute section and others in a relocatable section. The scope of each directive extends until a directive to the contrary is given. The Assembler initially starts in the relocatable section. Thus, if the first statement of a program were

```
A:  .ASECT
```

the label "A" would be a relocatable symbol which is assigned the value of relocatable zero. The absolute value of A will be calculated by the Linker by adding the value of the base of the relocatable section.

Example:

```
        .ASECT              ;ASSEMBLER IN ABSOLUTE SECTION
        .=1000              ;PC = 1000 ABSOLUTE
    A:  CLR X               ;A = 1000 ABSOLUTE
        .CSECT              ;ASSEMBLE IN RELOCATABLE SECTION
    X:  JMP A               ;X=0 RELOCATABLE
        .END
```

The absolute and/or relocatable section may be discontinued (by switching to the alternate section) and then continued where they left off by using another .ASECT or .CSBCT statement.

Example:

```
        .CSECT
        .WORD 0,1,2         ;ASSEMBLED AT RELOCATABLE 0, 2 and 4
        .ASECT
        .WORD 0,1,2         ;ASSEMBLED AT ABSOLUTE 0, 2 and 4
        .CSECT
        .WORD 0             ;ASSEMBLED AT RELOCATABLE 6.
        .END
```

If a label is defined twice, first in an absolute section and then in a relocatable section, the symbol will be relocatable but its value will be as defined in the absolute section.

1.8.4 .EOT

The .EOT directive indicates the physical End Of Tape though not the logical end of the program. If the .EOT is followed by a single line feed or form feed, the Assembler will still read to the end of the tape, but will not process anything past the .EOT directive. If .EOT is followed by at least two line feeds or form feeds, the Assembler will stop before the end of the tape. Either case is proper, but it should be understood that even though it appears as if the Assembler has read too far, it actually hasn't.

If a .EOT is embedded in a tape, and more information to be assembled follows it, .EOT must be immediately followed by at least two line feeds or form feeds. Otherwise, the first line following the .EOT will be lost.

Any operands following a .EOT directive will be ignored. The .EOT directive allows several physically separate tapes to be assembled as one program. The last tape should be terminated by a .END directive (see Section 1.8.6) but may be terminated with .EOT (see .END emulation in Section 1.9.4).

1.8.5 .EVEN

The .EVEN directive ensures that the assembly location counter is even by adding one if it is odd. Any operands following a .EVEN directive will be ignored.

1.8.6 .END

The .END directive indicates the logical and physical end of the source program. The .END directive may be followed by only one operand, an expression indicating the program's transfer address.

At load time, the load module will be loaded and program execution will begin at the transfer address indicated by the .END directive. If the address is not specified, the loader will halt after reading in the load module.

1.8.7 .WORD

The .WORD assembler directive may have one or more operands, separated by commas. Each operand is stored in a word of the object program. If there is more than one operand, they are stored in successive words. The operands may be any legally formed expression. For example,

```
        .=1420
        SAL=0
        .WORD 177535,.+4,SAL ;STORED IN WORDS 1420, 1422 AND
                             ;1424 WILL BE 177535, 1426, AND 0
```

Values exceeding 16 bits will be truncated from the left, to word length.

A .WORD directive followed by one or more void operands separated by commas will store zeros for the void operands. For example,

```
        .=1430                  ;ZERO, FIVE, AND ZERO ARE STORED
        .WORD ,5,               ;IN WORDS 1430, 1432, AND 1434
```

An operator field left blank will be interpreted as the .WORD directive if the operand field contains one or more expressions. 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 one of the logical operators, ! or &. For example,

```
        .=440                   ;THE OP-CODE FOR MOV, WHICH IS 010000,
        LABEL: +MOV,LABEL       ;IS STORED IN LOCATION 440. 440 IS
                                ;STORED IN LOCATION 442.
```

Note that the default .WORD will occur whenever there is a leading arithmetic or logical operator, or whenever a leading symbol is encountered which is not recognized as an instruction mnemonic or assembler directive. Therefore, if an instruction mnemonic 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 can result: A Q will occur because an expression operator is missing between MOR and A, and a U will occur if MOR is undefined. Two words will be generated; one for MOR A and one for B.

1.8.8 .BYTE

The .BYTE assembler directive may have one or more operands separated by commas. Each operand is stored in a byte of the object program. If multiple operands are specified, they are stored in successive bytes. The operands may be any legally formed expression with a result of 8 bits or less. For example,

```
        SAM=5                   ;STORED IN LOCATION 410 WILL BE
        .=410                   ;060 (THE OCTAL EQUIVALENT OF 48).
        .BYTE 48.,SAM           ;IN 411 WILL BE 005.
```

If the expression has a result of more than 8 bits, it will be truncated to its low-order 8 bits and will be flagged as a T error. If an operand after the .BYTE directive is left void, it will be interpreted as zero. For example,

```
        .=420                   ;ZERO WILL BE STORED IN
        .BYTE , ,               ;BYTES 420, 421 AND 422.
```

If the expression is relocatable, a warning flag, A, will be given.

1.8.9 .ASCII

The .ASCII directive translates strings of ASCII characters into their 7-bit ASCII codes with the exception of null, rubout, carriage return, line feed and form feed. The text to be translated is delimited by a character at the beginning and the end of the text. The delimiting character may be any printing ASCII character except colon and equal sign and those used in the text string. The 7-bit ASCII code generated for each character will be stored in successive bytes of the object program. For example,

```
.=500                    ;THE ASCII CODE FOR Y WILL BE
.ASCII /YES/             ;STORED IN 500, THE CODE FOR E
                         ;IN 501, THE CODE FOR S IN 502.
.ASCII /5+3/2/           ;THE DELIMITING CHARACTER OCCURS
                         ;AMONG THE OPERANDS. THE ASCII
                         ;CODES FOR 5 , + , AND 3 ARE
                         ;STORED IN BYTES 503, 504, AND
                         ;505. 2/ IS NOT ASSEMBLED.
```

The .ASCII directive may be terminated by any legal terminator except for = and :. Note that if the text delimiter is also a terminator, the leading text delimiter can also serve as the .ASCII directive terminator. For example,

```
.ASCII /ABCD/            ;THE SPACE IS REQUIRED
                         ;BECAUSE / IS NOT A TERMINATOR.
.ASCII+ABCD+             ;NO SPACE IS REQUIRED.
```

1.8.10 .RAD50

PDP-11 system programs often handle symbols in a specially coded form caled "RADIX 50" (this form is sometimes referred to as "MOD40"). This form allows 3 characters to be packed into 16 bits; therefore, any symbol can be held in two words, the form of the directive is:

```
.RAD50 /CCC/
```

The single operand is of the form /CCC/ where the slash (the delimiter) can be any printable character except for = and :. The delimiters enclose the characters to be converted which may be A through Z, 0 through 9, dollar ($), dot (.) and space ( ). If there are fewer than 3 characters they are considered to be left-justified and trailing spaces are assumed. Any characters following the trailing delimiter are ignored and no error results.

Examples:

```
.RAD50 /ABC/             ;PACK ABC INTO ONE WORD
.RAD50 /AB/              ;PACK AB (SPACE) INTO ONE WORD;
.RAD50 //                ;PACK 3 SPACES INTO ONE WORD
```

The packing algorithm is as follows:

A. Each character is translated into its RADIX 50 equivalent as indicated in the following table:

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

Note that another character can be defined for code 35.

B. The RADIX 50 equivalents for characters 1 through 3 (C1,C2,C3) are combined as follows:

```
RESULT=((C1*50)+C2)*50+C3
```

1.8.11 .LIMIT

A program often wishes to know the boundaries of the relocatable code. The .LIMIT directive generates 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 will always be even since all relocatable sections are loaded at even addresses and if a relocatable section consists of an odd number of bytes the linker adds one to the size to make it even.

1.8.12 Conditional Assembly Directives

Conditional assembly directives provide the programmer with the capability to conditionally include or not include portions of his source code in the assembly process. In what follows, E denotes an expression and S(i) denotes a symbol. The conditional directives are:

```
.IFZ      E          ;IF E=0
.IFNZ     E          ;IF E≠0
.IFL      E          ;IF E<0
.IFLE     E          ;IF E≤0
.IFG      E          ;IF E>0
.IFGE     E          ;IF E≥0
.IFDF     S (1) [!,&] S (2) [!,&]...[!,&] S(N)   (!=OR,&=AND)
.IFNDF    S (1) [!,&] S (2) [!,&]...[!,&] S(N)
```

If the condition is met, all statements up to the matching .ENDC are assembled. Otherwise, the statements are ignored until the matching .ENDC is detected.

In the above,.IFDF and .IFNDF mean "if defined" and "if undefined" respectively. The scan is left to right, no parentheses permitted.

Example:

.IFDF S!T&U	Means assemble if either S or T is defined and U is defined
.IFNDF T&U!S	Means assemble if both T and U are undefined or if S is undefined

General Remarks:

An errored or null expression takes the default value 0 for purposes of the conditional test. An error in syntax, e.g., a terminator other than ;, !, &, or CR results in the undefined situation for .IFDF and .IFNDF, as does an errored or null symbol.

All conditionals must end with the .ENDC directive. Anything in the operand field of .ENDC is ignored. Nesting is permitted up to a depth of 127 . Labels are permitted on conditional directives, but the scan is purely left to right. For example:

```
        .IFZ 1
A:      .ENDC
```

A is ignored.

```
A:      .IFZ 1
        .ENDC
```

A is entered in the symbol table.

If a .END is encountered while inside a satisfied conditional, a Q flag will appear, but the .END directive will still be processed normally. If more .ENDC's appear than are required, Q flags appear on the extras.

1.9 OPERATING PROCEDURES

1.9.1 Introduction

The Assembler enables you to assemble an ASCII tape containing PAL-11 statements into a relocatable binary tape (object module). To do this, two or three passes are necessary. On the first pass, the Assembler creates a table of user-defined symbols and their associated values, and a list of undefined symbols is printed on the teleprinter. On the second pass the Assembler assembles the program and punches out an absolute binary tape and/or outputs an assembly listing. During the third pass (this pass is optional), the Assembler punches an absolute binary tape or outputs an assembly listing. The symbol table (and/or a list of errors) may be output on any of these passes. The input and output devices as well as various options are specified during the initial dialogue (see Section 1.9.3). The Assembler initiates the dialogue immediately after being loaded and after the last pass of an assembly.

1.9.2 Loading PAL-11S

PAL-11S is loaded by the Paper Tape Software Absolute Loader. Note that on systems with hardware switch registers, the start address of the Absolute Loader must be in the Switch Register when loading the Assembler. This is because the Assembler tape has an initial program which clears all of core up to the address specified in the Switch Register, and jumps to that address to start loading the Assembler.

1.9.3 Initial Dialogue

After being loaded, the Assembler prints its name and version and then initiates dialogue by printing on the teleprinter

```
*S
```

meaning "What is the Source symbolic input device?" The response may be

↵	use Low-speed reader (↵denotes typing the RETURN key)
H	meaning High-speed reader
L	meaning Low-speed reader
T	meaning Teleprinter keyboard

The device specification is terminated, as is all user response, by typing the RETURN key.

If an error is made in typing at any time, typing the RUBOUT key will erase the immediately preceding character if it is on the current line. Typing CTRL/U will erase the whole line on which it occurs.

After the *S question and response, the Assembler prints:

*B

meaning "What is the Binary output device?" The responses to *B are similar to those for *S:

H	meaning High-speed punch
L	meaning Low-speed punch
↵	meaning do not output binary tape (↵denotes typing the RETURN key)

In addition to I/O device specification, various options may be chosen. The binary output will occur on the second pass unless /3 (indicating the third pass) is typed following the H or L. Errors will be listed on the same pass if /E is typed. If /E is typed in response to more than one inquiry, only the last occurrence will be honored. It is strongly suggested that the errors be listed on the same pass as the binary output, since errors may vary from pass to pass.

If both /3 and /E are typed, /3 must precede /E. The response is terminated by typing the RETURN key. Examples:

*B	L/E	Binary output on the low-speed punch and the errors on the teleprinter, both during the second pass.
*B	H/3/E	Binary output on the high-speed punch and the errors on the teleprinter during the third pass.
*B	↵	The RETURN key alone will cause the Assembler to omit binary output

After the *B question and response, the Assembler prints:

*L

meaning "What is the assembly Listing output device?" The response to *L may be:

L	meaning Low-speed punch
H	meaning High-speed punch
T	meaning Teleprinter
P	meaning Line Printer
↵	meaning do not output listing (↵denotes typing RETURN)

After the I/O device specification, pass and error list options similar to those for *B may be chosen. The assembly listing will be output on the third pass unless /2 (indicating the second pass) is typed following H, L, T, or P. Errors will be listed on the teleprinter during the same pass if /E is typed. If both /2 and /E are typed, /2 must precede /E. The response is terminated by typing the RETURN key. Examples:

*L	L/2/E	Listing on low-speed punch and errors on teleprinter during second pass.
*L	H	Listing on high-speed punch during third pass
*L	↵	The RETURN key alone will cause the Assembler to omit listing output.

After the *L question and response, the final question is printed on the teleprinter:

```
        *T
```

meaning "What is the symbol Table output device?" The device specification is the same as for *L question. The symbol table will be output at the end of the first pass unless /2 or /3 is typed in response to *T. The first tape to be assembled should be placed in the reader before typing the RETURN key because assembly will begin upon typing RETURN to the *T question. The /E option is not a meaningful response to *T. Example

*T	T/3	Symbol table output on teleprinter at end of third pass.
*T		Typing the RETURN key alone will cause the Assembler to omit symbol table output.

The symbol table is printed alphabetically, three symbols per line. Each symbol printed is followed by its identifying characters and by its value. If the symbol is undefined, six asterisks replace its value. The identifying characters indicate the class of the symbol; that is, whether it is a label, direct assignment, register symbol, etc. The following examples show the various forms.

```
ABCDEF          001244      (Defined Label)
R3         =   %000003      (Register Symbol)
DIRASM     =    177777      (Direct Assignment)
XYZ        =    ******      (Undefined direct assignment)
R6         =   %******      (Undefined register symbol)
LABEL      =    ******      (Undefined label)
```

Generally, undefined symbols and external symbols will be listed as undefined direct assignments. Multiply-defined symbols are not flagged in the symbol table printout but are flagged wherever they are used in the program.

If the symbol is relocatable or global or both, the symbol's value will be followed by an R, a G or both.

It is possible to output both the binary tape and the assembly listing on the same pass, thereby reducing the assembly process to two passes (see Example 1 below). This will happen automatically unless the binary device and the listing device are conflicting devices or the same device (see Example 2 below). The only conflicting devices are the teleprinter and the low-speed punch. Even though the Assembler deduces that three passes are necessary, the binary and listing can be forced on pass 2 by including /2 in the responses to *B and *L (see Example 3 below).

Example 1. Runs 2 passes:

```
    *S   H    High-speed reader
    *B   H    High-speed punch
    *L   P    Line Printer
    *T   T    Teleprinter
```

Example 2. Runs 3 passes:

```
    *S   H    High-speed reader
    *B   H    High-speed punch
    *L   H    High-speed punch
    *T   T    Teleprinter
```

Example 3. Runs 2 passes:

```
     *S   H    High-speed reader
     *B   H/2  High-speed punch on pass 2
     *L   H/2  High-speed punch on pass 2
     *T   T    Teleprinter
```

Note that there are several cases where the binary output can be intermixed with ASCII output:

```
a.   *B     H/2        Binary and listing to punch on pass 2.
     *L     H/2

b.   *B     L/E        Binary to low-speed punch and error  listing  to
                       teleprinter (and low-speed punch).

c.   *B     L/2/E      Binary, error listing, and
     *L     T/2        listing to low speed punch.
```

The object module so generated is acceptable to the Linker as long as there are no CTRL/A characters in the source program. The start of every block on the binary tape is indicated by a 001 and the Linker ignores all information until a 001 is detected. Thus, all source and/or error messages will be ignored if they do not contain any CTRL/A characters (octal 001).

If a character other than those mentioned is typed in reponse to a question, the Assembler will ignore it and print the question again. Example:

```
     *S   H    High-speed reader
     *B   Q    Q is not a valid response
     *B        The question is repeated
```

If at any time you wish to restart the Assembler, type CTRL/P. If the low-speed reader is the source input device, turn it off before typing CTRL/P.

When no passes are omitted or error options specified, the Assembler performs as follows:

PASS 1:

Assembler creates a table of user-defined symbols and their associated values to be used in assembling the source to object program. Undefined symbols (not including external globals) are listed on the teleprinter at the end of the pass. The symbol table is also listed at this time. If an illegal location statement of the form .=expression is encountered, the line and error code will be printed out on the teleprinter before the assembly proceeds. An error in a location statement is usually a fatal error in the program and should be corrected.

PASS 2:

Assembler punches the object module, and prints the pass error count and undefined location statements on the teleprinter.

PASS 3:

Assembler prints or punches the assembly program listing, undefined location statements, and the pass error count on the teleprinter.

The functions of passes 2 and 3 will occur simultaneously on pass 2 if the binary and listing devices are different, and do not conflict with each other (the low-speed punch and teleprinter conflict). Furthermore, if the binary object module is not requested, the listing will be produced on pass 2.

The following table summarizes the initial dialogue questions:

PRINTOUT	INQUIRY
*S	What is the input device of the Source symbolic tape?
*B	What is the output device of the Binary object tape?
*L	What is the output device of the assembly Listing?
*T	What is the output device of the symbol Table?

The following table summarizes the legal responses:

CHARACTER	RESPONSE INDICATED
T	Teleprinter keyboard
L	Low-speed reader or punch
H	High-speed reader or punch
P	Line Printer
/1	Pass 1
/2	Pass 2
/3	Pass 3
/E	Errors listed on same pass (not meaningful response to *S or *T)
↵	Omit function (except in response to *S).

Typical examples of complete initial dialogues:

For minimal PDP-11 configuration:

*S	L	Source input on low-speed reader
*B	L/E	Binary output on low-speed punch errors during same (second) pass
*L	T	Listing on teleprinter during pass 3
*T	T	Symbol table on teleprinter at end of pass 1

For a PDP-11 with high-speed I/O devices:

*S	H	Source input on high-speed reader
*B	H/E	Binary output on high-speed punch errors during same (second) pass
*L	↵	No listing
*T	T/2	Symbol table on teleprinter at end of pass 2.

1.9.4 Assembly Dialogue

During assembly, the Assembler will pause to print on the teleprinter various messages to indicate that you must respond in some way before the assembly process can continue. You may also type CTRL/P, at any time, if you wish to stop the assembly process and restart the initial dialogue, as mentioned in the previous section.

When a .EOT assembler directive is read on the tape, the Assembler prints

EOF ?

and pauses. During this pause, the next tape is placed in the reader, and RETURN is typed to continue the assembly.

If the specified assembly listing output device is the high-speed punch and if it is out of tape, or if the device is the Line Printer and is out of paper, the Assembler prints on the teleprinter

```
EOM ?
```

and waits for tape or paper to be placed in the device. Type the RETURN key when the tape or paper has been replenished; assembly will continue.

Conditions causing the EOM ? messages for an assembly listing device are:

HSP	LPT
No power	No power
No tape	Printer drum gate open
	Too hot
	No paper

There is no EOM if the line printer is switched off-line, although characters may be lost for this condition as well as for an EOM.

If the binary output device is the high-speed punch and if it is out of tape, the Assembler prints:

```
EOM ?
*S
```

The assembly process is aborted and the initial dialogue is begun again.

When a .END assembler directive is read on the tape, the Assembler prints:

```
END ?
```

and pauses. During the pause the first tape is placed in the reader, and the RETURN key is typed to begin the next pass. On the last pass, the .END directive causes the Assembler to begin the initial dialogue for the next assembly.

If you are starting the binary pass and the binary is to be punched on the low-speed punch, turn the punch on before typing the RETURN key for starting the pass. The carriage return and line feed characters will be punched onto the binary tape, but the Linker will ignore them.

If the last tape ends with a .EOT, the Assembler may be told to emulate a .END assembler directive by responding with E followed by the RETURN key. The Assembler will then print

```
END ?
```

and wait for another RETURN before starting the next pass. Example:

```
EOF ?  E↵
END ?
```

Note that forcing a .END in this manner causes the error counter to be incremented by one.

1.9.5 Assembly Listing

PAL-11S produces a side-by-side assembly listing of symbolic source statements, their octal equivalents, assigned addresses, and error codes, as follows:

```
EELLLLLL OOOOOOASSS.......S
         OOOOOO
         OOOOOO
```

The E's represent the error field. The L's represent the address. The O's represent the object data in octal. The S's represent the source statement. "A" represents a single apostrophe which indicates that either the second, third or both words of the instruction will be modified by the Linker. While the Assembler accepts 72 characters per line on input, the listing is reduced by the 16 characters to the left of the source statement.

The above represents a three-word statement. The second and third words of the statement are listed under the command word. No addresses precede the second and third words since the address order is sequential.

The third line is omitted for a two-word statement; both second and third lines are omitted for a one-word statement.

For a .BYTE directive, the object data field is three octal digits.

For a direct assignment statement, the value of the defining expression is given in the object code field although it is not actually part of the code of the object program.

The .ASECT and .CSECT directives cause the current value of the appropriate location counter (absolute or relocatable) to be printed.

Each page of the listing is headed by a page number (octal).

1.9.6 Object Module Output

The output of the assembler during the binary object pass is an object module which is meaningful only to the linker. What follows gives an overview of what the object module contains and at what stage each part of it is produced.

The binary object module consists of three main types of data block:

a) Global symbol directory	(GSD)
b) Text blocks	(TXT)
c) Relocation Directory	(RLD)

1.9.6.1 Global Symbol Directory - As the name suggests, the GSD contains a list of all the global symbols together with the name of the object module. Each symbol is in Radix-50 form and contains information regarding its mode and value whenever known.

The GSD is created at the start of the binary object pass.

1.9.6.2 Text Block - The text blocks consist entirely of the binary object data as shown in the listing. The operands are in the unmodified form.

1.9.6.3 Relocation Directory - The RLD blocks consist of directives to the Linker which may reference the text block preceding the RLD. These directives control the relocation and linking process.

Text and RLD blocks are constructed during the binary object pass. Outputting of each block is done whenever either the TXT or RLD buffer is full and whenever the location counter needs to be modified.

1.10 ERROR CODES

The error codes printed beside the octal and symbolic code in the assembly listing have the following meanings:

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.
I	Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing.
L	Line buffer overflow. Extra characters on a line (more than 72) 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.
P	Phase error. A label's definition or 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.
S	Symbol table overflow. When the quantity of user-defined symbols exceeds the allocated space available in the user's symbol table, the assembler outputs the current source line with the S error code, then returns to the initial dialogue.

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.

1.11 SOFTWARE ERROR HALTS

PAL-11S loads all of its unused trap vectors with the code

```
.WORD .+2,HALT
```

so that if the trap does occur, the processor will halt in the second word of the vector. The address of the halt, displayed in the console address register, therefore indicates the cause of the halt.

Address of Halt (octal)	Meaning
12	Reserved instruction executed
16	Trace trap occurred
26	Power fail trap
32	EMT executed

A halt at address 40 indicates an IOXLPT detected error. R0 (displayed in the console lights) contains an identifying code:

Code in R0	Meaning
0	Illegal memory reference, SP overflow or illegal instruction.
1	Illegal IOX command.
2	Slot number out of range.
3	Device number illegal
4	Referenced slot not INITed.
5	Illegal Data Mode.

IOXLPT also sets R1 as follows:

If the error code is 0, R1 contains the PC at the time of the error.

If the error code is 1-5, R1 points to some element in the IOT argument list or to the instruction following the argument list, depending on whether IOXLPT has finished decoding all the arguments when it detects the error.

CHAPTER 2

WRITING PAL-11A ASSEMBLY LANGUAGE PROGRAMS[1]

PAL-11A (Program Assembly Language for the PDP-11's Absolute Assembler) enables you to write source (symbolic) programs using letters, numbers, and symbols which are meaningful to you. The source programs, generated either on-line using the Text Editor (ED-11), or off-line, are then assembled into object programs (in absolute binary) which are executable by the computer. The object program is produced after two passes through the Assembler; an optional third pass produces a complete octal/symbolic listing of the assembled program. This listing is especially useful for documentation and debugging purposes.

This chapter explains not only how to write PAL-11A programs but also how to assemble the source programs into computer-acceptable object programs. All facets of the assembly language are explained and illustrated with many examples, and the chapter concludes with assembling procedures. In explaining how to write PAL-11A source programs it is necessary, especially at the outset, to make frequent forward references. Therefore, we recommend that you first read through the entire chapter to get a "feel" for the language, and then reread the chapter, this time referring to appropriate sections as indicated, for a thorough understanding of the language and assembling procedures.

Some notable features of PAL-11A are:

1. Selective assembly pass functions

2. Device specification for pass functions

3. Optional error listing on Teletype

4. Double buffered and concurrent I/O (provided by IOX)

5. Alphabetized, formatted symbol table listing

The PAL-11A Assembler is available in two versions: a 4K version and an 8K version.

The assembly language applies equally to both versions. The 4K version provides symbol storage for about 176 user-defined symbols, and the 8K version provides for about 1256 user-defined symbols (see Section 2.3).

In addition, the 8K version allows a line printer to be used for the program listing and/or symbol table listing.

[1] PAL-11A is not currently available for PDP-11 systems without switch registers.

The following discussion of the PAL-11A Assembly Language assumes that you have read the PDP-11 Processor Handbook, with emphasis on those sections which deal with the PDP-11 instruction set, formats, and timings -- a thorough knowledge of these is vital to efficient assembly language programming.

2.1 CHARACTER SET

A PAL-11A source program is composed of symbols, numbers, expressions, symbolic instructions, assembler directives, arguments separators, and line terminators written using the following ASCII[1] characters.

1. The letters A through Z. (Upper and lower case letters are acceptable, although upon input, lower case letters will be converted to upper case letters.)
2. The numbers 0 through 9.
3. The characters . and $ (reserved for system software).
4. The separating or terminating symbols:

   : = % # @ ( ) , ; " ' + - & !

   carriage return   tab   space   line feed   form feed

2.2 STATEMENTS

A source program is composed of a sequence of statements, where each statement is on a single line. The statement is terminated by a carriage return character and must be immediately followed by either a line feed or form feed character. Should a carriage return character be present and not be followed by a line feed or form feed, the Assembler will generate a Q error (Section 2.10) and that portion of the line following the carriage return will be ignored. Since the carriage return is a required statement terminator, a line feed or form feed not immediately preceded by a carriage return will have one inserted by the Assembler.

It should be noted that, if the Editor (ED-11) is being used to create the source program (see Section 4.4.4), a typed carriage return (RETURN key) automatically generates a line feed character.

A statement may be composed of up to four fields which are identified by their order of appearance and by specified terminating characters as explained below and summarized in Appendix B. The four fields are:

Label   Operator   Operand   Comment

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.

ASCII stands for American Standard Code for Information Interchange.

2.2.1 Label

A label is a user-defined symbol (see Section 3.3.2) which is assigned the value of the current location counter. It 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 100(octal), the statement

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

will assign the value 100(octal) to the label ABCD so that subsequent reference to ABCD will be to location 100(octal). More than one label may appear within a single label field; each label within the field will have the same value. For example, if the current location is 100, multiple labels in the statement

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

will equate each of the three labels ABC, $DD, and A7.7 with the value 100(octal). ($ and . are reserved for system software.)

The error code M (multiple definition of a symbol) will be generated during assembly if two or more labels have the same first six characters.

2.2.2 Operator

An operator follows the label field in a statement, and may be an instruction mnemonic or an assembler directive (see Appendix B). When it is an instruction mnemonic, it specifies what action is to be performed on any operand(s) which follows it. When it is an assembler directive, it specifies a certain function or action to be performed during assembly.

The operator may be preceded only by one or more labels and may be followed by one or more operands and/or a comment. An operator is legally terminated by a space, tab, or any of the following characters.

```
        #  +  -  @  (  "  '  %  !  &  ,  ;
        line feed      form feed      carriage return
```

The use of each character above will be explained in this chapter.

Consider the following examples:

```
        MOV  A,B        ;⇥(TAB) terminates operator MOV
        MOV@A,B         ;@ terminates operator MOV
```

When the operator stands alone without an operand or comment, it is terminated by a carriage return followed by a line feed or form feed character.

2.2.3 Operand

An operand is that part of a statement which is operated on by the operator -- an instruction mnemonic or assembler directive. Operands may be symbols, expressions, or numbers. When multiple operands appear within a statement, each is separated from the next by a comma.

An operand may be preceded by an operator and/or label, and followed by a comment.

The operand field is terminated by a semicolon when followed by a comment, or by a carriage return followed by a line feed or form feed character when the operand ends the statement. For example,

```
        LABEL:     MOV GEORGE,BOB      ;THIS IS A COMMENT
```

where the space between MOV and GEORGE terminated the operator field and began the operand field; the comma separated the operands GEORGE and BOB; the semicolon terminated the operand field and began the comment.

2.2.4 Comments

The comment field is optional and may contain any ASCII character except null, rubout, carriage return, line feed or form feed. All other characters, even those with special significance are ignored by Assembler when used in the comment field.

The comment field may be preceded by none, any, or all of the other three fields. It must begin with the semicolon and end with a carraige return followed by a line feed or form feed character. For example,

```
        LABEL:    CLR HERE      ;THIS IS A $1.00 COMMENT
```

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

2.2.5 Format Control

The format is controlled by the space and tab characters. They have no effect on the assembling process of the source program unless they are embedded within a symbol, number, or ASCII text; or are used as the operator field terminator. Thus, they can be used to provide a neat, readable 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 much easier to read.

Page size is controlled by the form feed character. A page of n lines is created by inserting a form feed (CTRL/FORM keys on the keyboard) after the nth line. If no form feed is present, a page is terminated after 56 lines.

2.3 SYMBOLS

There are two types of symbols, permanent and user-defined. Both are stored in the Assembler's symbol table. Initially, the symbol table contains the permanent symbols, but as the source program is assembled, user-defined symbols are added to the table.

2.3.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics (see Appendix B.3) and assembler directives (see Section 2.8). These symbols are a permanent part of the Assembler's symbol table and need not be defined before being used in the source program.

2.3.2 User-Defined Symbols

User-defined symbols are those defined as labels (see Section 2.2.1) or by direct assignment (see Section 2.3.3). These symbols are added to the symbol table as they are encountered during the first pass of the assembly. They can be composed of alphanumeric characters, dollar signs, and periods only; again, dollar signs and periods are reserved for use by the system software. Any other character is illegal and, if used, will result in the error message I (see Section 2.11). The following rules also apply to user-defined symbols:

1. The first character must not be a number.

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

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

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

A user-defined symbol may duplicate a permanent symbol. The value associated with a permanent symbol that is also user-defined depends upon its use:

1. A permanent symbol encountered in the operator field is associated with its corresponding machine op-code.

2. If a permanent symbol in the operand field is also user-defined, its user-defined value is associated with the symbol. If the symbol is not found to be user-defined, then the corresponding machine op-code value is associated with the symbol.

2.3.3 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 Assembler's 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 newly assigned value will replace the previous value assigned to the symbol.

The general format for a direct assignment statement is

symbol = expression

The following conventions apply:

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

2. A direct assignment statement may be preceded by a label and may be followed by a comment.

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

4. Only one level of forward referencing is allowed.

Example of the two levels of forward referencing (illegal):

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

X and Y are both undefined throughout pass 1 and will be listed on the printer as such at the end of that pass. X is undefined throughout pass 2, and will cause a U error message.

Examples:

```
A = 1               ;THE SYMBOL A IS EQUATED WITH THE VALUE 1

B = 'A-1&MASKLOW    ;THE SYMBOL B IS EQUATED WITH THE EXPRES-
                    ;SION'S VALUE.

C:   D = 3          ;THE SYMBOL D IS EQUATED WITH 3.  THE
E:   MOV #1,ABLE    ;LABELS C AND E ARE EQUATED WITH THE
                    ;NUMERICAL MEMORY ADDRESS OF THE MOV
                    ;COMMAND.
```

2.3.4 Register Symbols

The eight general registers of the PDP-11 are numbered 0 through 7. These registers may be referenced by use of a register symbol, that is, a symbolic name for a register. A register symbol is defined by means of a direct assignment, where the defining expression contains at least one term preceded by a % or at least one term previously defined as a register symbol.

```
R0=%0           ;DEFINE R0 AS REGISTER 0
R3=R0+3         ;DEFINE R3 AS REGISTER 3
R4=1+%3         ;DEFINE R4 AS REGISTER 4
THERE=%2        ;DEFINE "THERE" AS REGISTER 2
```

It is important to note that all register symbols must be defined before they are referenced. A forward reference to a register symbol will generally cause phase errors (see Section 2.10).

The % may be used in any expression thereby indicating a reference to a register. Such an expression is a register expression. Thus, the statement

```
        CLR %6
```

will clear register 6 while the statement

```
        CLR 6
```

will clear the word at memory address 6. 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 and are thoroughly explained in Sections 2.6 and 2.7. Two obvious examples of this are:

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

        CLR    X(2)        ;ANY EXPRESSION ENCLOSED IN
                           ;() MUST BE A REGISTER.  IN
                           ;THIS CASE, INDEX REGISTER 2.
```

2.4 EXPRESSIONS

Arithmetic and logical operators (see Section 2.4.2) may be used to form expressions. A term of an expression may be a permanent or user-defined symbol, a number, ASCII data, or the present value of the assembly location counter represented by the period. Expressions are evaluated from left to right. Parenthetical grouping is not allowed.

Expressions are evaluated as word quantities. The operands of a .BYTE directive (Section 2.8.5) are evaluated as word expressions before truncation to the low-order eight bits.

A missing term or expression will be interpreted as 0. A missing operator will be interpreted as +. The error code Q (Questionable syntax) will be generated for a missing operator. For example,

```
        A +    -100     ;OPERAND MISSING
```

will be evaluated as A + 0 - 100, and

```
        TAG ! LA 177777 ;OPERATOR MISSING
```

will be evaluated as TAG ! LA+177777.

2.4.1 Numbers

The Assembler accepts both octal and decimal numbers. Octal numbers consist of the digits 0 through 7 only. Decimal numbers consist of the digits 0 through 9 followed by a decimal point. If a number contains an 8 or 9 and is not followed by a decimal point, the N error code (see Section 2.10) will be printed and the number interpreted as decimal. Negative numbers may be expressed as a number preceded by a minus sign rather than in a two's complement form. Positive numbers may be preceded by a plus sign although this is not required.

If a number is too large to fit into 16 bits, the number is truncated from the left. In the assembly listing the statement will be flagged with a Truncation (T) error.

2.4.2 Arithmetic and Logical Operators

The arithmetic operators are:

+	indicates addition or a positive number
-	indicates subtraction or a negative number

The logical operators are defined and illustrated below.

&	indicates the logical AND operation
!	indicates the logical inclusive OR operation

AND
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1

OR
0 ! 0 = 0
0 ! 1 = 1
1 ! 0 = 1
1 ! 1 = 1

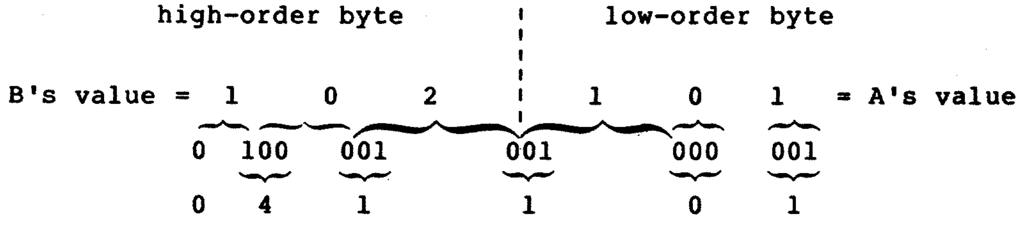
2.4.3 ASCII Conversion

When preceded by an apostrophe, any ASCII character (except null, rubout, carriage return, line feed, or form feed) is assigned the 7-bit ASCII value of the character (see Appendix A). For example,

```
'A
```

is assigned the value 101(octal).

When preceded by a quotation mark, two ASCII characters (not including null, rubout, carriage return, line feed, or form feed) are assigned the 7-bit ASCII values of each of the characters to be used. Each 7-bit value is stored in an 8-bit byte and the bytes are combined to form a word. For example, "AB will store the ASCII value of A in the low-order (even) byte and the value of B in the high-order (odd) byte:



"AB = 041101

2.5 ASSEMBLY LOCATION COUNTER

The period (.) is the symbol for the assembly location counter. (Note difference of Program Counter. . = PC. See Section 2.6.) When used in the operand field of an instruction, it represents the address of the first word of the instruction. When used in the operand field of an assembler directive, it represents the address of the current byte or word. For example,

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

(# is explained in Section 2.6.9).

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
```

The expression defining the period must not contain forward references or symbols that vary from one pass to another. Examples:

```
        .=500

FIRST:  MOV  .+10,COUNT ;THE LABEL FIRST HAS THE VALUE(OCTAL)
                        ;.+10 EQUALS 510(OCTAL).  THE CONTENTS
                        ;OF THE LOCATION 510(OCTAL) WILL BE DE-
                        ;POSITED IN LOCATION COUNT.

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

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

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

```
        .=.+100
```

will reserve 100(octal) bytes of storage space in the program. The next instruction will be stored at 1100.

2.6 ADDRESSING

The Program Counter (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 (see Section 2.6.11), the action of the Program Counter must be understood. The key rule is:

Whenever the processor implicitly uses the Program Counter (PC) 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 (see Sections 2.6.7, 2.6.8, and 2.6.10), 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 Section 3.4.

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 address field as described in the PDP-11 Handbook.

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

2.6.1 Register Mode

The register contains the operand.

Format: R

Example:

```
        R0 = %0         ;DEFINE R0 AS REGISTER 0

        CLR  R0         ;CLEAR REGISTER 0
```

2.6.2 Deferred Register Mode

The register contains the address of the operand.

Format: @R or (ER)

Example:

```
        CLR @R1         ;CLEAR THE WORD AT THE
           or           ;ADDRESS CONTAINED IN
        CLR (1)         ;REGISTER 1.
```

2.6.3 Autoincrement Mode

The contents of the register are incremented immediately after being used as the address of the operand.[1]

[1] a. Both JMP and JSR instructions using mode 2 may increment the register before or after its use, depending on what PDP-11 processor is being used. This mode should be avoided.
b. In double operand instructions of the addressing form %R, (R)+ or %R,-(R) where the source and destination registers are the same, the results may be different when executed on different PDP-11 processors. The use of these forms should be avoided!

Format: (ER)+

Examples:

```
        CLR   (R0)+       ;CLEAR WORDS AT ADDRESSES
        CLR   (R0+3)+     ;CONTAINED IN REGISTERS 0, 3, AND 2 AND
        CLR   (2)+        ;INCREMENT REGISTER CONTENTS
                          ;BY TWO.
```

2.6.4 Deferred Autoincrement Mode

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

Format: @(ER)+

Example

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

2.6.5 Autodecrement Mode

The contents of the register are decremented before being used as the address of the operand.[1]

Format: -(ER)

Examples:

```
        CLR -(R0)         ;DECREMENT CONTENTS OF REG-
        CLR -(R0+3)       ;ISTERS 0, 3, AND 2 BEFORE USING
        CLR -(2)          ;AS ADDRESSES OF WORDS TO BE CLEARED
```

2.6.6 Deferred Autodecrement Mode

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

Format: @-(ER)

Example:

```
        CLR   @-(2)       ;DECREMENT CONTENTS OF REG. 2
                          ;BEFORE USING AS POINTER TO ADDRESS
                          ;OF WORD TO BE CLEARED
```

2.6.7 Index Mode

Format: E(ER)

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.

[1] See previous footnote.

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
```

2.6.8 Deferred Index Mode

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

Format: @E(ER)

Example:

```
        CLR @14(4)      ;IF REGISTER 4 HOLDS 100, AND LOCA-
                        ;TION 114 HOLDS 2000, LOC.  2000 IS
                        ;CLEARED
```

2.6.9 Immediate Mode and Deferred Immediate (Absolute) 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: #E

Examples:

```
        MOV  #100, R0   ;MOVE AN OCTAL 100 TO REGISTER 0

        MOV  #X, R0     ;MOVE THE VALUE OF SYMBOL X TO
                        ;REGISTER 0
```

The operation of this mode is explained as follows:

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

```
0 1 2 7 0 3

0 0 0 1 0 0
```

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

If the #E is preceded by @, E specifies an absolute address.

2.6.10 Relative and Deferred Relative Modes

Relative Mode is the normal mode for memory references.

Format E

Examples:

```
        CLR  100        ;CLEAR LOCATION 100

        MOV  X,Y        ;MOVE CONTENTS OF LOCATION X TO
                        ;LOCATION Y
```

This mode is assembled as Index Mode, using 7, the PC, as the 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. Rather, it is the number which, when added to the PC, becomes the address of the operand. Thus, the base is X - PC. The operation is explained as follows.

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

```
    Location 20:        0 1 6 7 0 3

    Location 22:        0 0 0 0 5 4
```

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

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

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

This mode is called relative because the operand address is calculated relative to the current PC. The base is the distance (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.

If E is preceded by @, the expression's value is the pointer to the address of the operand.

2.6.11 Table of Mode Forms and Codes (6-bit (A) format only - see Section 3.7)

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 forms however, increases the instruction length by one word.

	Form	Mode	Meaning
None of these forms increase the instruction length.	R	0n	Register
	@R or (ER)	1n	Register n deferred
	(ER)+	2n	Autoincrement
	@(ER)+	3n	Autoincrement deferred
	-(ER)	4N	Autodecrement
	@-(ER)	5N	Autodecrement deferred

Any of these forms adds a word to the instruction length	E(ER)	6n	Index
	@E(ER)	7n	Index deferred
	#E	27	Immediate
	@#E	37	Absolute memory reference
	E	67	Relative
	@E	77	Relative deferred reference

Notes:

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

2. The form @#E differs from the form E in that the second or third word of the instruction contains the absolute address of the operand rather than the relative distance between the operand and the PC. Thus, the statement CLR @#100 will clear location 100 even if the instruction is moved from the point at which it was assembled.

2.7 INSTRUCTION FORMS

The instruction mnemonics are given in Appendix B. This section defines the number and nature of the operand fields for these instructions.

In the table that follows, let R, E, and ER represent expressions as defined in Section 3.4, and let A be a 6-bit address specification of the forms:

```
E     @E
R     @R  or  (R)
(ER)+ @(ER)+
-(ER) @-(ER)
E(ER) @E(ER)
#E    @#E
```

Table 2-1
Instruction Operand Fields

Instruction	Form	Example
Double Operand	Op A,A	MOV (R6)+,@Y
Single Operand	Op A	CLR -(R2)
Operate	Op	HALT
Branch	Op E where -128 ≤(E-.-2)/2≤127	BR X+2 BLO .-4
Subroutine Call	JSR ER,A	JSR PC,SUBR
Subroutine Return	RTS ER	RTS PC
EMT/TRAP	Op or Op E where 0≤E≤377(octal)	EMT EMT 31

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.

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(octal)) it is truncated to eight bits and a Truncation (T) error occurs.

2.8 ASSEMBLER DIRECTIVES

Assembler directives (sometimes called pseudo-ops direct the assembly process and may generate data. They may be preceded by a label and followed by a comment. The assembler directive occupies the operator field. Only one directive may be placed in any one statement. One or more operands may occupy the operand field or it may be void -- allowable operands vary from directive to directive.

2.8.1 .EOT

The .EOT directive indicates the physical End-of-Tape though not the logical end of the program. If the .EOT is followed by a single line feed or form feed, the Assembler will still read to the end of the tape, but will not process anything past the .EOT directive. If .EOT is followed by at least two line feeds or form feeds, the Assembler will stop before the end of the tape. Either case is proper, but it should be understood that even though it appears as if the Assembler has read too far, it actually hasn't.

If a .EOT is embedded in a tape, and more information to be assembled follows it, .EOT must be immediately followed by at least two line feeds or form feeds. Otherwise, the first line following the .EOT will be lost.

Any operands following a .EOT directive will be ignored. The .EOT directive allows several physically separate tapes to be assembled as one program. The last tape is normally terminated by a .END directive (see Section 3.8.3) but may be terminated with .EOT (see .END emulation in Section 3.9.4).

2.8.2 .EVEN

The .EVEN directive ensures that the assembly location counter is even by adding one if it is odd. Any operands following a .EVEN directive will be ignored.

2.8.3 .END

The .END directive indicates the logical and physical end of the source program. The .END directive may be followed by only one operand, an expression indicating the program's entry point.

At load time, the object tape will be loaded and program execution will begin at the entry point indicated by the .END directive. If the entry point is not specified, the Loader will halt after reading in the object tape.

2.8.4 .WORD

The .WORD assembler directive may have one or more operands, separated by commas. Each operand is stored in a word of the object program. If there is more than one operand, they are stored in successive words. The operands may be any legally formed expressions. For example,

```
        .=1420
        SAL=0
        .WORD 177535,.+4,SAL  ;STORED IN WORDS 1420, 1422, AND
                              ;1424 WILL BE 177535, 1426, AND 0.
```

Values exceeding 16 bits will be truncated from the left, to word length.

A .WORD directive followed by one or more void operands separated by commas will store zeros for the void operands. For example,

```
        .=1430                ;ZERO, FIVE, AND ZERO ARE STORED
        .WORD ,5,             ;IN WORDS 1430, 1432, AND 1434.
```

An operator field left blank will be interpreted as the .WORD directive if the operand field contains one or more expressions. The first term of the first expression in the operand field must not be an instruction or assembler directive unless preceded by a +, -, or one of the logical operators ! or &. For example,

```
        .=440                 ;THE OP-CODE FOR MOV, WHICH IS 010000,
        LABEL:  +MOV,LABEL    ;IS STORED IN LOCATION 440.  440 IS
                              ;STORED IN LOCATION 442.
```

Note that the default .WORD will occur whenever there is a leading arithmetic or logical operator, or whenever a leading symbol is encountered which is not recognized as an instruction mnemonic or assembler directive. Therefore, if an instruction mnemonic 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 can result: a Q will occur because an expression operator is missing between MOR and A, and a U will occur if MOR is undefined. Two words will be generated; one for MOR A and one for B.

2.8.5 .BYTE

The .BYTE assembler directive may have one or more operands separated by commas. Each operand is stored in a byte of the object program. If multiple operands are specified, they are stored in successive bytes. The operands may be any legally formed expression with a result of 8 bits or less. For example,

```
        SAM=5           ;STORED IN LOCATION 410 WILL BE
        .=410           ;060 (THE OCTAL EQUIVALENT OF 48).
        .BYTE 48.,SAM   ;IN 411 WILL BE 005.
```

If the expression has a result of more than 8 bits, it will be truncated to its low-order 8 bits and will be flagged as a T error. If an operand after the .BYTE directive is left void, it will be interpreted as zero. For example,

```
        .=420           ;ZERO WILL BE STORED IN
        .BYTE , ,       ;BYTES 420, 421 AND 422.
```

2.8.6 .ASCII

The .ASCII directive translates strings of ASCII characters into their 7-bit ASCII codes with the exception of null, rubout, carriage return, line feed, and form feed. The text to be translated is delimited by a character at the beginning and the end of the text. The delimiting character may be any printing ASCII character except colon and equal sign and those used in the text string. The 7-bit ASCII code generated for each character will be stored in successive bytes of the object program. For example,

```
        .=500           ;THE ASCII CODE FOR "Y" WILL BE
        .ASCII /YES/    ;STORED IN 500, THE CODE FOR "E"
                        ;IN 501, THE CODE FOR "S" IN 502.

        .ASCII /5+3/2/  ;THE DELIMITING CHARACTER OCCURS
                        ;AMONG THE OPERANDS. THE ASCII
                        ;CODES FOR "5", "+", AND "3" ARE
                        ;STORED IN BYTES 503, 504, AND
                        ;505. 2/ IS NOT ASSEMBLED.
```

The ASCII directive must be terminated by a space or a tab.

2.9 OPERATING PROCEDURES

2.9.1 Introduction

The Assembler enables you to assemble an ASCII tape containing PAL-11A statements into an absolute binary tape. To do this, two or three passes are necessary. On the first pass the Assembler creates a table of user-defined symbols and their associated values, and a list of undefined symbols is printed on the teleprinter. On the second pass the Assembler assembles the program and punches out an absolute binary tape and/or outputs an assembly listing. During the third pass (this pass is optional) the Assembler punches an absolute binary tape or outputs an assembly listing. The symbol table (and/or a list of errors) may be output on any of these passes. The input and output devices as well as various options are specified during the initial dialogue (see Section 3.3.9). The Assembler initiates the dialogue immediately after being loaded and after the last pass of an assembly.

2.9.2 Loading PAL-11A

PAL-11A is loaded by the Absolute Loader (see Chapter 6 for operating procedures). Note that the start address of the Absolute Loader must be in the Switch Register when loading the Assembler. This is because the Assembler tape has an initial portion which clears all of core up to the address specified in the Switch Register, and jumps to that address to start loading the Assembler.

2.9.3 Initial Dialogue

After being loaded, the Assembler initiates dialogue by printing on the teleprinter:

```
*S
```

meaning "What is the Source symbolic input device?" The response may be:

H	meaning High-speed reader
L	meaning Low-speed reader
T	meaing Teletype keyboard

If the response is T, the source program must be typed at the terminal once for each pass of the assembly and it must be identical each time it is typed.

The device specification is terminated, as is all user response, by typing the RETURN key.

If an error is made in typing at any time, typing the RUBOUT key will erase the immediately preceding character if it is on the current line. Typing CTRL/U will erase the whole line on which it occurs.

After the *S question and response, the Assembler prints:

```
*B
```

meaning "What is the Binary output device?" The responses to *B are similar to those for *S:

H	meaning High-speed punch
L	meaning Low-speed punch
↵	meaning do not output binary tape (↵ denotes typing the RETURN key)

In addition to I/O device specification, various options may be chosen. The binary output will occur on the second pass unless /3 (indicating the third pass) is typed following the H or L. Errors will be listed on the same pass if /E is typed. If /E is typed in response to more than one inquiry, only the last occurrence will be honored. It is strongly suggested that the errors be listed on the same pass as the binary output, since errors may vary from pass to pass. If both /3 and /E are typed, /3 must precede /E. The response is terminated by typing the RETURN key. Examples:

*B L/E — Binary output on the low-speed punch and the errors on the the teleprinter, both during the second pass.

*B H/3/E — Binary output on the high-speed punch and the errors on the teleprinter, both during the third pass.

*B ↵ — Typing just the RETURN key will cause the Assembler to omit binary output.

After the *B question and response, the Assembler prints:

```
*L
```

meaning "What is the assembly Listing output device?" The response to *L may be:

L   meaning Low-speed punch (outputs a tab as a tab-rubout)

H   meaning High-speed punch

T   meaning Teleprinter (outputs a tab as multiple spaces)

P   meaning line Printer (8K version only)

↵   meaning do not output listing
(↵ denotes typing the RETURN key)

After the I/O device specification, pass and error list options similar to those for *B may be chosen. The assembly listing will be output to the third pass unless /2 (indicating the second pass) is typed following H, L, T, or P. Errors will be listed on the teleprinter during the same pass if /E is typed. If both /2 and /E are typed, /2 must precede /E. The response is terminated by typing the RETURN key. Examples:

*L L/2/E — Listing on low-speed punch and errors on teleprinter during second pass.

*L H — Listing on high-speed punch during third pass.

*L ↵ — The RETURN key alone will cause the Assembler to omit listing output.

After the *L question and response, the final question is printed on the teleprinter:

```
*T
```

meaning "What is the symbol Table output device?" The device specification is the same as for the *L question. The symbol table will be output at the end of the first pass unless /2 or /3 is typed in response to *T. The first tape to be assembled should be placed in the reader before typing the RETURN key because assembly will begin upon typing the RETURN key in response to the *T question. The /E option is not a meaningful response to *T. Example:

*T T/3 — Symbol table output on teleprinter at end of third pass.

*T ↵ — Typing just the RETURN key will cause the Assembler to omit the symbol table output.

The symbol table is printed alphabetically, four symbols per line. Each symbol printed is followed by its identifying characters and by its value. If the symbol is undefined, six asterisks replace its value. The identifying characters indicate the class of the symbol; that is, whether it is a label, direct-assignment, register symbol, etc. The following examples show the various forms:

```
ABCDEF          001244          (Defined label)

R3        =     %000003         (Register symbol)

DIRASM    =     177777          (Direct assignment)

XYZ       =     ******          (Undefined direct assignment)

R6        =     %******         (Undefined register symbol)

LABEL     =     ******          (Undefined label)
```

Generally, undefined symbols (including labels) will be listed as undefined direct assignments.

Multiply-defined symbols are not flagged in the symbol table printout but they are flagged wherever they are used in the program.

It is possible to output both the binary tape and the assembly listing on the same pass, thereby reducing the assembly process to two passes (see Example 1 below). This will happen automatically unless the binary device and the listing device are conflicting devices or the same device (see Example 2 below). The only conflicting devices are the teleprinter and the low-speed punch. Even though the Assembler deduces that three passes are necessary, the binary and listing can be forced on pass 2 by including /2 in the responses to *B and *L (see Example 3 below).

Example 1. Runs 2 passes:

```
*S  H            High-speed reader

*B  H            High-speed punch

*L  P            Line Printer

*T  T            Teleprinter
```

Example 2. Runs 3 passes:

```
*S  H            High-speed reader

*B  H            High-speed punch

*L  H            High-speed punch

*T  T            Teleprinter
```

Example 3. Runs 2 passes:

```
*S  H            High-speed reader

*B  H/2          High-speed punch on pass 2

*L  H/2          High-speed punch on pass 2

*T  T            Teleprinter
```

Note that there are several cases where the binary output can be intermixed with ASCII output:

a.	*B H/2 *L H/2	Binary and listing to punch on pass 2
b.	*B L/E	Binary to low-speed punch and error listing to teleprinter (and low-speed punch)
c.	*B L/2/E *L T/2	Binary, error listing, and listing to low-speed punch.

The binary so generated is loadable by the Absolute Loader as long as there are no CTRL/A characters in the source program. The start of every block on the binary tape is indicated by a 001 and the Absolute Loader ignores all information until a 001 is detected. Thus, all source and/or error messages will be ignored if they do not contain any CTRL/A characters (octal 001).

If a character other than those mentioned is typed in response to a question, the Assembler will ignore it and print the question again. Example:

*S H	High-speed reader
*B Q	Q is not a valid response
*B	The question is repeated

If at any time you wish to restart the Assembler, type CTRL/P.

When no passes are omitted or error options specified, the Assembler performs as follows:

PASS 1: Assembler creates a table of user-defined symbols and their associated values to be used in assembling the source to object program. Undefined symbols are listed on the teleprinter at the end of the pass. The symbol table is also listed at this time. If an illegal location statement of the form .=expression is encountered, the line and error code will be printed out on the teleprinter before the assembly proceeds. An error in a location statement is usually a fatal error in the program and should be corrected.

PASS 2: Assembler punches the object tape, and prints the pass error count and undefined location statements on the teleprinter.

PASS 3: Assembler prints or punches the assembly program listing, undefined location statements, and the pass error count on the teleprinter.

The functions of passes 2 and 3 will occur simultaneously on pass 2 if the binary and listing devices are different, and do not conflict with each other (low-speed punch and Teleprinter conflict).

The following table summarizes the initial dialogue questions:

Printout	Inquiry
*S	What is the input device of the Source symbolic tape?
*B	What is the output device of the Binary object tape?
*L	What is the output device of the assembly Listing?
*T	What is the output device of the symbol Table?

The following table summarizes the legal responses:

Character	Response Indicated
T	Teletype keyboard or printer
L	Low-speed reader or punch
H	High-speed reader or punch
P	Line Printer (8K version only)
/1	Pass 1
/2	Pass 2
/3	Pass 3
/E	Errors listed on same pass (not meaningful in response to *S or *T)
↵	Omit function

Typical examples of complete initial dialogues:

For minimal PDP-11 configuration:

*S	L	Source input on low-speed reader
*B	L/E	Binary output on low-speed punch Errors during same (second) pass
*L	T	Listing on teleprinter during pass 3
*T	T	Symbol table on teleprinter at end of pass 1

For a PDP-11 with high-speed I/O devices:

*S	H	Source input on high-speed reader
*B	H/E	Binary output on high-speed punch, Errors during same (second) pass.
*L		No listing
*T	T/2	Symbol table on teleprinter at end of pass 2

2.9.4 Assembly Dialogue

During assembly, the Assembler will pause to print on the teleprinter various messages to indicate that you must respond in some way before the assembly process can continue. You may also type CTLR/P, at any time, if you wish to stop the assembly process and restart the initial dialogue, as mentioned in the previous section.

When a .EOT assembler directive is read on the tape, the assembler prints:

```
EOF ?
```

and pauses. During this pause, the next tape is placed in the reader, and RETURN is typed to continue the assembly.

If the specified assembly listing output device is the high-speed punch and if it is out of tape, or if the device is the Line Printer and is out of paper, the Assembler prints on the teleprinter:

```
EOM ?
```

and waits for tape or paper to be placed in the device. Type the RETURN key when the tape or paper has been replenished; assembly wil continue.

Conditions causing the EOM? message for an assembly listing device are:

HSP	LPT
No power	No power
No tape	Printer drum gate open
	Too hot
	No paper

There is no EOM if the line printer is switched off-line, although characters may be lost for this condition as well as for an EOM. If the binary output device is the high-speed punch and if it is out of tape, the Assembler prints:

```
EOM ?
*S
```

The assembly process is aborted and the initial dialogue is begun again.

When a .END assembler directive is read on the tape, the Assembler prints:

```
END ?
```

and pauses. During the pause the first tape is placed in the reader, and the RETURN key is typed to begin the next pass. On the last pass, the .END directive causes the Assembler to begin the initial dialogue for the next assembly.

If you are starting the binary pass and the binary is to be punched on the low-speed punch, turn the punch on before typing the RETURN key for starting the pass. The carriage return and line feed characters will be punched onto the binary tape, but the Absolute Loader will ignore them.

If the last tape ends with a .EOT, the Assembler may be told to emulate a .END assembler directive by responding with E followed by the RETURN key. The Assembler will then print:

```
END ?
```

and wait for another RETURN before starting the next pass. Example:

```
EOF ?    E
END ?
```

NOTE

> When a .END directive is emulated with an E response to the EOF? message, the error counter is incremented.
>
> To avoid incrementing the error counter, place a paper tape containing only the line .END in the reader and press the RETURN key instead of using the E response.

2.9.5 Assembly Listing

PAL-11A produces a side-by-side assembly listing of symbolic source statements, their octal equivalents, assigned absolute addresses, and error codes as follows:

```
EELLLLLL 000000 SSS......S
         000000
         000000
```

The E's represent the error field. The L's represent the absolute address. The O's represent the object data in octal. The S's represent the source statement. While the Assembler accepts 72(decimal) characters per line on input, the listing is reduced by the 16 characters to the left of the source statement.

The above represents a three-word statement. The second and third words of the statement are listed under the command word. No addresses precede the second and third word since the address order is sequential.

The third line is omitted for a two-word statement; both second and third lines are omitted for a one-word statement.

For a .BYTE directive, the object data field is three octal digits.

For a direct assignment statement, the value of the defining expression is given in the object code field although it is not actually part of the code of the object program.

Each page of the listing is headed by a page number.

2.10 ERROR CODES

The error codes printed beside the octal and symbolic code in the assembly listing have the following meanings:

Error Code Meaning

A Addressing error. An address within the instruciton is incorrect.

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 then once.

I Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing.

L Line buffer overflow. Extra characters on a line (more than 72(decimal)) 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 no decimal point.

P Phase error. A label's definition or 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.

S Symbol table overflow. When the quantity of user-defined symbols exceeds the allocated space available in the user's symbol table, the assembler outputs the current source line with the S error code, then returns to the initial dialogue.

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.

2.11 SOFTWARE ERROR HALTS

PAL-11A loads all unused trap vectors with the code

```
        .WORD   .+2,HALT
```

so that if the trap does occur, the processor will halt in the second word of the vector. The address of the halt, displayed in the console address register, therefore indicates the cause of the halt. In addition to the halts which may occur in the vectors, the standard IOX error halt at location 40 may occur (see Chapter 7).

Address of Halt	Meaning
12	Reserved instruction executed
16	Trace trap occurred
26	Power fail trap
32	EMT executed
40	IOX detected error

See Appendix B for summaries of PAL-11A features.

CHAPTER 3

LINK-11S LINKER

3.1 INTRODUCTION

3.1.1 General Description

LINK-11S (stand alone) is a PDP-11 system program designed to link and relocate programs previously assembled by PAL-11S. The user can separately assemble the main program and each of its various subroutines without assigning an absolute load address at assembly time. The binary output of assembly (called an object module) is processed by LINK-11S to:

1. Relocate each object module and assign absolute addresses.

2. Link the modules by correlating global symbols defined in one module and referenced in other modules.

3. Print a load map which displays the assigned absolute addresses.

4. Punch a load module which can subsequently be loaded (by the Absolute Loader) and executed.

Some of the advantages of using PAL-11S and LINK-11S are:

1. The program is divided into segments (usually subroutines) which are assembled separately. If an error is discovered in one segment, only that segment needs to be reassembled. The new object module is then linked with the other object modules.

2. Absolute addresses need not be assigned at assembly time. The Linker automatically assigns absolute addresses. This keeps programs from overlaying each other. This also allows subroutines to change size without influencing the placement of other routines.

3. Separate assemblies allow the total number of symbols to exceed the number allowed in a single assembly.

4. Internal symbols (symbols which are not global) need not be unique among object modules. Thus, naming rules are required only for global symbols when separate programmers prepare separate subroutines of a single program.

5. Subroutines may be provided for general use in object module form to be linked into the user's program.

LINK-11S is designed to run on an 8K PDP-11 with an ASR-33. A PC11 (high speed paper tape reader and punch) and an LP11 (line printer) may be used if available. The PC11 significantly speeds up the linking process. An LP11 provides a fast device for the load map listing.

3.1.2 Absolute and Relocatable Program Sections

A program assembled by PAL-11S may consist of an absolute program section, declared by the .ASECT assembler directive, and a relocatable program section, declared by the .CSECT assembler directive. (If a program has neither an .ASECT or .CSECT directive, the assembler implicitly assumes a .CSECT directive.) The program and data in the absolute section are assigned absolute addresses as specified by the location counter setting statements (.=x). The Linker assigns absolute addresses to the program and data in the relocatable section. Addressses are normally assigned such that the relocatable section is at the high end of memory. The assignment of addresses may be influenced by command string options (see Section 3.3.2).

The Linker appropriately modifies all instructions and/or data as necessary to account for the relocation of the control section.

LINK-11S can handle object modules containing named control (relocatable) sections as generated by PAL-11R. However, PAL-11S can create only the unnamed control section (which has the special default name of 6 blanks) and the absolute section (with the special name . ABS.). The unnamed control section is internal to each object module. That is, every object module may have an unnamed control section (each with the name 6 blanks) but the Linker treats them independently. Each is assigned an absolute address such that they occupy mutually exclusive areas of memory. Named control sections, on the other hand, are treated globally. That is, if different object modules each have control sections with the same name, they are all assigned the same absolute load address and the size of the area reserved or loading of the section is the maximum of the sizes of each section. Thus, named control sections allow the sharing of data and/or programs among object modules. This is very similar to the handling and function of labelled COMMON in FORTRAN IV. A restriction of LINK-11S is that the name of a control section must not be the same as the name of a global entry symbol, as this results in multiple definition errors.

3.1.3 Global Symbols

Global symbols provide the links for communication between object modules (or assemblies). Global symbols are created with the .GLOBL assembler directive. Symbols which are not global are called internal symbols. If the global symbol is defined (as a label or direct assignment) in an object module it is called an entry symbol, and other object modules may reference it. If the global symbol is not defined in the object module it is an external symbol. It is assumed to be defined (as an entry symbol) in some other object module.

As the Linker reads the object modules it records all the global symbol definitions and references. It then modifies the instructions and/or data that reference the global symbols.

3.2 INPUT AND OUTPUT

3.2.1 Object Module

Input to LINK-11S is the object module. This is the output of PAL-11S (or any other program which can create an object module). The Linker reads each object module twice; that is, it is a two-pass processor.

On pass 1, the Linker reads each object module to gather enough information to assign absolute addresses to all relocatable sections and absolute values to all globals. This information appears in the global symbol directory (GSD) of the object module.

On pass 2, the Linker reads all of each object module and produces the load module (see Section 3.2.2). The data gathered on pass 1 guides the relocation and linking process on pass 2.

3.2.2 Load Modules

The normal output of the Linker is a load module which may be loaded and run.

A load module consists of formatted binary blocks holding absolute load addresses and object data as specified for the Paper Tape System Absolute Loader and the PDP-11 Disk Monitor. The first few words of data are the communications directory (COMD) and have an absolute load address equal to the lowest relocated address of the program. The absolute loader loads the COMD at the specified address but the program subsequently overlays it.[1] The disk monitor loader expects the COMD and loads it where the monitor wants it. The end of the load module is indicated by a TRA block; that is, a block containing only a load address. The byte count in the formatted binary block is 6 on this block; on all other blocks the byte count is larger than 6. The TRA (transfer address) is selected by the Linker to be the first even transfer address seen. Thus, if four object modules are linked together and if the first and second had a .END statement, the third had a .END A and the fourth had a .END B, the transfer address would be A of module three.

[1] The overlaying of the COMD by the relocated program is a trick to allow the Absolute Loader to handle load modules with a COMD. However, a problem arises if a load module is to be loaded by the absolute loader and either of the following conditions exists:

a. The object modules used to construct the load module contained no relocatable code; or

b. The total size of the relocatable code is less than 20 (decimal) bytes (the size of the COMD).

In either case, there is not enough relocatable code to overlay the COMD which means the COMD will load into parts of memory not intended to be altered by the user. The COMD's load address, selected by the Linker in the above cases, is such that it will be up against the current top of memory (see *T option in section 3.3.1). If the top happens to be very low, the Linker does not allow the COMD to be loaded below address 0; it loads it at 0.

3.2.3 Load Map

The load map provides several types of information concerning the load module's make-up. The map begins with an indication of the low and high limits of the relocatable code and the transfer address. Then there is a section of the map for each object module included in the linking process. Each of these sections begins with the module name followed by a list of the control sections and the entry points for each control section. For each control section, the base of the section (its low address) and its size (in bytes) is printed to the right of the section name (enclosed in angle brackets). Following each section name is a list of entry points and their addresses. After all information has been printed for each object module, any undefined symbols are listed. Note that modules are loaded such that if modules A, B and C are linked together, A is lowest and C is highest in memory.

The format is quite self-explanatory as can be seen from the following example:

```
LOAD MAP

TRANSFER ADDRESS: 037434
LOW LIMIT: 037406
HIGH LIMIT: 037460
**********
MODULE  MOD1
SECTION ENTRY   ADDRESS SIZE
<. ABS.>        000000  000000
<      >        037406  000044
        X3      037452
        X4      037440
        X5      037450
        X7      037430
**********
MODULE  MOD2
SECTION ENTRY   ADDRESS SIZE
<      >        037452  000006
        X1      037452
        X2      037452
**********
**********
UNDEFINED REFERENCES
X6

PASS 2

*
```

3.3 OPERATING PROCEDURES

3.3.1 Loading and Command String

The Linker is loaded by the Absolute Loader and is self-starting. It uses a simple command dialogue which allows the object module, load module and load map devices to be specified. During pass 1 and pass 2, the Linker asks for each object module individually.

Operation begins by the linker typing its name and version. This is followed by the input option printed as *IΔ. The responses are:

↵	Read object module from HSR.
H↵	Read object module from HSR.
L↵	Read object module from LSR.

The input option is followed by the output option *OΔ. The responses are:

↵	Punch load module on HSP.
H↵	Punch load module on HSP.
L↵	Punch load module on LSP.

LINK-11S asks if a load map is desired by typing *MΔ. The legal responses are ↵ for no map, T↵ or H↵ or P↵for a map on the teleprinter, high-speed punch, or line printer, respectively.

The next two options concern the placement of the relocated object program in memory. The standard version of the Linker assumes it is linking for an 8K machine. It relocates the program such that it is as high as possible in 8K but leaves room for the Absolute and Boot Loaders. These assumed values may be changed by altering parameters HGHMEM (highest legal memory address +1) and ALODSZ (number of bytes allocated for Absolute Loader and Boot Loader) and reassembling the Linker. The user may control where a program is relocated to with the *T and *B options. After the option *TΔ has been typed, the user may respond as follows:

↵	Relocate so that program is up against the current top of memory. If the top has not been changed, then the top is the assembled-in top (HGHMEM-ALODSZ). The standard assumption is 16272 decimal (16384-112) or 37460 octal.
N↵	N is an octal number (unsigned) which defines a new top address.

If a new top is specified, the *B option is suppressed.

After the option *BΔ has been printed the user may respond as follows:

↵	Use current top of memory.
N↵	N is an unsigned octal number which defines the bottom address of the program. That is, a new top of memory is calculated so that the bottom of the program corresponds with N.

Once a top of memory has been calculated (by *T or *B), that value is used until it is changed.

LINK-11S indicates the start of pass one by typing PASS 1. The input is requested by the Linker, one tape at a time, by typing *Δ. The legal responses are:

↵ Read a tape and request more input.

U↵ List all undefined globals on the teleprinter and request more input.

E↵ End of input. If there are undefined globals, list them on the teleprinter and request more input. Otherwise print the load map, if requested, and enter pass 2.

C↵ End of input. Assign 0 to any undefined globals, print the load map (if requested), and enter pass 2.

The Linker indicates the start of pass 2 by typing PASS 2. It then requests each input tape as in pass 1.

A carriage return is the only useful response to * on pass 2. The modules must be read on pass 2 in the same order as pass 1. When the last module has been read the Linker automatically finishes the load module and restarts itself.

Leader and trailer are punched on the load module.

If the LSP is being used for the load module output, it should be turned on before pass 2 begins. Thus, turn it on before typing E↵or C↵ The echo of these characters (and the load map, if printed on the TTY) is punched on the load module but may be easily removed since leader is punched on the load module. In any case, ASCII information in a load module is ignored by the Absolute and Disk Monitor loaders. However, the LSP can be turned on while leader is being punched (after the linker has typed PASS 2) to keep the load map, etc., from being punched onto the tape.

Note:

On all command string options, except for *T and *B, the linker examines only the last character typed preceding the carriage return. Thus,

ABCDEFGH↵

is equivalent to H↵

3.3.1.1 Operational Cautions - The Linker does not give a warning if a program is linked so low in memory that it goes below address 0. However, this case is easily seen by examining the low and high limits which are always printed (on the load map or on the teleprinter).

The Linker reads object modules until an end of medium is detected. Object modules from the DEC Program Library contain a special checksum at the end of the tape which must be removed before they are linked. Failure to remove this checksum can result in fatal Linker errors.

3.3.2 Error Procedure and Messages

3.3.2.1 Restarting - CTRL/P (symbolized as ^P) is used for two purposes by LINK-11. If a ^P is typed while a load map is being printed, the load map is aborted and the Linker continues. A ^P typed at any other time causes the Linker to restart itself.

3.3.2.2 Non-Fatal Errors -

1. Non-unique object module name - this error is detected during pass 1; an error message is issued and the module is rejected. The message is:

   ```
   ?MODULE NAME xxxxxx NOT UNIQUE
   ```

   The Linker then asks for more input.

2. Load map device EOM - this error allows the user to fix the device and continue or abort the map listing. The Linker prints:

   ```
   ?MAP DEVICE EOM.
   TYPE <CR> TO CONTINUE
   ```

   Any response, terminated by ↵ or ↓ causes the Linker to continue. A ↑ P causes the map to be aborted.

3. A byte relocation error - the Linker tries to relocate and link byte quantities. However, relocation usually fails and linking may fail. Failure is defined as the high byte of the relocated value (or the linked value) not being all zero. In such a case, the value is truncated to 8 bits and the following message is printed:

   ```
   ?BYTE RELOC ERROR AT ABS ADDRESS xxxxxx.
   ```

   The Linker automatically continues.

4. If the object modules are not read in the same order on pass 2 as pass 1, the Linker indicates which module should be loaded next by typing:

   ```
   ?LOAD xxxxxx NEXT!
   ```

   The linker then asks for more input.

5. Multiply-Defined Globals - this results in the following error message during pass 1:

   ```
   ?xxxxxx MULTIPLY DEFINED BY MODULE xxxxxx.
   ```

   The second definition is ignored and the Linker continues.

3.3.2.3 Fatal Errors - Each of the following errors causes the indicated error message to be printed and the Linker to be restarted.

1. Symbol Table overflow - the message is:

   ```
   ?SYMBOL TABLE OVERFLOW - MODULE xxxxxx, SYMBOL xxxxxx
   ```

2. System Errors - this class of errors prints:

```
?SYSTEM ERROR xx
```

where xx is an identifying number as follows:

Number	Meaning
01	Unrecognized symbol table entry was found.
02	A relocation directory references a global name which cannot be found in the symbol table.
03	A relocation directory contains a location counter modification command which is not last.
04	Object module does not start with a GSD.
05	The first entry in the GSD is not the module name.
06	An RLD references a section name which cannot be found.
07	The TRA specification references a non-existent module name.
08	The TRA specification references a non-existent section name.
09	An internal jump table index is out of range.
10	A checksum error occurred on the object module.
11	An object module binary block is too big (more than 64 decimal words of data).
12	A device error occurred on the load module output device.

All system errors except for numbers 10 and 12 indicate a program failure either in the Linker or the program which generated the object module. Error 05 can occur if a tape is read which is not an object module.

3.3.2.4 <u>Error HALTs</u> - LINK-11S loads all of its unused trap vectors with the code:

```
        .WORD     .+2, HALT
```

so that if the trap occurs, the processor halts in the second word of the vector. The address of the halt, displayed in the console lights, therefore indicates the cause of the halt.

Address of HALT (octal)	Meaning
12	Reserved instruction executed.
16	Trace trap occurred.
26	Power fail trap.
32	EMT executed.

A halt at address 40 indicates an IOXLPT detected error. R0 (displayed in the console lights) contains an identifying code:

Code in R0	Meaning
0	Illegal memory reference, SP overflow or illegal instruction.
1	Illegal IOX command.
2	Slot number out of range.
3	Device number illegal.
4	Referenced slot not INIT ed.
5	Illegal data mode.

IOXLPT also sets Rl as follows:

If the error code is 0, Rl contains the PC at the time of the error.

If the error code is 1-5, Rl points to some element in the IOT argument list or to the instruction following the argument list, depending on whether IOXLPT has finished decoding all the arguments when it detects the error.

CHAPTER 4

EDITING THE SOURCE PROGRAM

The PDP-11 Text Editor program (ED-11) enables you to display your source program (or any text) on the teleprinter, make corrections or additions to it, and punch all or any portion of the program on paper tape. This is accomplished by typing simple one-character commands on the keyboard.

The Editor commands can be grouped according to function:

1. input/output;

2. searching for strings of characters;

3. positioning the current character location pointer;

4. inserting, deleting, and exchanging text portions.

All input/output functions are handled by IOX, the PDP-11 Input/Output Executive (see Chapter 7).

4.1 COMMAND MODE AND TEXT MODE

Whenever ED-11 prints an * on the teleprinter, you may type a command to it. (Only one command per line is acceptable.) The Editor is then said to be in Command Mode. While most commands operate exclusively in this mode, there are five ED-11 commands that require additional information in order for the commands to be carried out. The Editor goes into Text Mode to receive this text.

Should a nonexistent command be typed or a command appear in incorrect format, ED-11 prints a ?. This is followed by an * at the beginning of a new line indicating that the Editor is in Command Mode.

Editor processing begins in Command Mode. When you type a command, no action occurs until you follow it by typing the RETURN key (symbolized as ↵). If the command is not a text-type command, typing the RETURN key initiates the execution of the command and ED-11 remains in Command Mode. However, if the command is a text-type command (Insert, eXchange, Change, Get, or wHole), typing the RETURN key causes the Editor to to into Text Mode. At this time you should type the text to be operated on by the command. This can include the non-printing characters discussed below, as well as spaces and tabs (up to eight spaces generated by the CTRL/TAB keys).

Note that typing the RETURN key always causes the physical return of the Teletype print element to the beginning of the line, and automatically generates a line feed, thereby advancing the carriage to a new line. In Text Mode, the RETURN key not only serves these mechanical functions, allowing you to continue typing at the beginning of a new line, but at the same time it enters a carriage return and line feed character into the text. (A carriage return not followed by a line feed cannot, therefore, be entered from the keyboard.)

RETURN and LINE FEED are both counted as characters and can be edited along with the printing characters (as can the form feed, discussed in Section 4.2.5). When you wish to terminate Text Mode and reenter Command Mode, you must type the LINE FEED key symbolized as ↓). A typed LINE FEED is not considered to be part of the text unless it is the first character entered in Text Mode.

4.2 COMMAND DELIMITERS

4.2.1 Arguments

Some ED-11 commands require an argument to specify the particular portion of text to be affected by the command or how many times to perform the command. In other commands this specification is implicit and arguments are not allowed.

The ED-11 command arguments are described as follows:

1. n stands for any number from 1 through 32767 (decimal) and may, except where noted, be preceded by a + or -.

   If no sign precedes n, n is assumed to be a positive number.

   Where an argument is acceptable, its absence implies an argument of 1 (or -1 if a - is present).

   The role of n varies according to the command with which it is associated.

2. 0 refers to the beginning of the current line.

3. @ refers to a marked (designated) character location (see Section 4.2.3).

4. / refers to the end of text in the Page Buffer.

The roles of all arguments are explained further with the corresponding commands which qualify them.

4.2.2 The Character Location Pointer (Dot)

Almost all ED-11 commands function with respect to a movable reference point, Dot. This character pointer is normally located between the most recent character operated upon and the next character and, at any given time, can be thought of as "where the Editor is" in your text. There are commands which move Dot anywhere in the text, thereby redefining the "current location" and allowing greater facility in the use of the other commands.

4.2.3 Mark

In addition to Dot, a secondary character pointer known as Mark also exists in ED-11. This less agile pointer is used with great effect to mark or "remember" a location by moving to Dot and conditionally remaining there while Dot moves on to some other place in the text. Thus, it is possible to think of Dot as "here" and Mark as "there". Positioning of Mark, which is referenced by means of the argument @, is discussed below in several commands.

4.2.4 Line-Oriented Command Properties

ED-11 recognizes a line as a unit by detecting a line terminator in the text. This means that ends of lines (line feed or form feed characters) are counted in line-oriented commands. This is important to know, particularly if Dot, which is a character location pointer, is not pointing at the first character of a line.

In such a case, an argument n does not affect the same number of lines (forward) as its negative (backward). For example, the argument -1 applies to the character string beginning with the first character following the second previous end-of-line character and ending at Dot; argument +1 applies to the character string beginning at Dot and ending at the first end-of-line character. If Dot is located, say, in the center of a line, notice that this affects 1-1/2 lines back or 1/2 line forward, respectively:

Example of List Commands -1L and +1L:

```
            Text                Command         Printout

     CMPB   ICHAR,#033          *-1L            BEQ $ALT
     BEQ    $ALT                                CMPB I
     CMPB   ICHAR, #175         *+1L
     BNE    PLACE                               CHAR,#175    Dot remains
            Dot is here                                      here
```

4.2.5 The Page Buffer

The Page Buffer holds the text being edited. The unit of source data that is read into the Page Buffer from a paper tape, is the page. Normally a page is terminated, and therefore defined, by a form feed (CTRL/FORM) in the source text wherever a page is desired. (A form feed is an acceptable Text Mode character.) Overflow, no-tape, or reader-off conditions can also end a page of input (as described in Section 4.3.1.2). Since more than one page of text can be in the buffer at the same time, it should be noted that the entire contents of the Page Buffer are available for editing.

4.3 COMMANDS

4.3.1 Input and Output Commands

Three commands are available for reading in a page of text. The Read command (Section 4.3.1.2) is a specialized input command; the Next command (Section 4.3.1.4) reads in a page after punching out the previous page; and the wHole command (Section 4.3.3.2) reads in and punches out pages of text as part of a search for a specified character string.

Output commands either list text or punch it on paper tape. The List command causes specified lines of text to be printed at the terminal so that they may be examined. Paper tape commands (Next and wHole also perform input) provide for the output of specified pages, lines, form feeds (for changing the amount of data that constitutes a given page), and blank tape. Note that the process of outputting text does not cause Dot to move.

4.3.1.1 Open - The Open command (O) should be typed whenever a new tape is put in the reader. This is used when the text file being edited is on more than one paper tape.

Note also that if the reader is off at the time an input command is given, turning the reader on must be followed by the Open command.

4.3.1.2 Read - One way of getting a page of text into the Page Buffer so that it can be edited is the Read (R) command. The R command causes a page of text to be read from either the low-speed reader or high-speed reader (as specified in the starting dialogue, Section 4.4.2), and appended to the contents (if any) of the Page Buffer.

Text is read in until either:

1. A form feed character is encountered;

2. The page buffer is 128 characters from being filled, or a line feed is encountered after the buffer has become 500 characters from being filled;

3. The reader is turned off, or runs out of paper tape (see Open command, Section 4.3.1.1).

Following execution of an R command, Dot and Mark are located at the beginning of the Page Buffer.

A 4K system can accommodate about 4000 characters of text. Each additional 4K of memory provides space for about 8000 characters.

NOTE

> An attempt to overflow the storage area causes the command (in this case, R) to stop executing. A ? is then printed, followed by an * on the next line indicating that a command may be typed. No data is lost.

4.3.1.3 List and Punch - Output commands List (L) and Punch (P) can be described together, as they differ only in that the device addressed by the former is the terminal, and the device addressed by the latter is the paper tape punch. Dot is not moved by these commands.

nL	Lists	the character string beginning at Dot and ending with the nth end-of-line
nP	Punches	
-nL	Lists	the character string beginning with the first character following the (n+1)th previous end-of-line and terminating at Dot
-nP	Punches	
0L	Lists	the character string beginning with the first character of the current line and ending at Dot
0P	Punches	
@L	Lists	the character string between Dot and the Marked location
@P	Punches	
/L	Lists	the character string beginning at Dot and ending with the last character in the Page Buffer
/P	Punches	

In addition to the above List commands, there are three special List commands that accept no arguments. The current line is defined as the line containing Dot, i.e., from the line feed (or form feed) preceding Dot to the line feed (or form feed) following Dot.

V	Lists the entire line containing Dot
<	Same as -1L. If Dot is located at the beginning of a line, this simply lists the line preceding the current line
>	Lists the line following the current line

Examples:

```
              TEXT                 COMMANDS          PRINTOUT

        CMPB   ICHAR,#033             V          CMPB   ICHAR,#175
        BEQ    $ALT                   <          BEQ    $ALT
        CMPB   ICHAR,#175                        CMPB   I
        BNE    PLACE                  >          BNE    PLACE

Dot is here.                                   Dot remains here.
```

4.3.1.4 Next - Typing nN punches out the entire contents of the Page Buffer (followed by a trailer of blank tape if a form feed is the last character in the buffer), deletes the contents of the buffer, and reads the Next page into the buffer. It performs this sequence n times. If there are fewer than the n pages specified, the command is executed for the number of pages actually available, and a ? is printed out. Following execution of a Next, Dot and Mark are located at the beginning of the Page Buffer.

4.3.1.5 Form Feed and Trailer -

F Punches out a Form feed character and four inches of blank tape

nT Punches out four inches of Trailer (blank) tape n times

4.3.1.6 Procedure with Low-Speed Punch - If the low speed punch is the specified output device (see Section 4.4.2), the Editor pauses before executing any tape command just typed (Punch, Form feed, Trailer, Next, wHole). The punch must be turned on at this time, after which typing the SPACE bar initiates the execution of the command. Following completion of the operation, the Editor pauses again to let you turn the punch off. When the punch has been turned off, typing the SPACE bar returns ED-11 to Command Mode.

4.3.2 Commands to Move Dot and Mark

4.3.2.1 Beginning and End -

B Moves Dot to the Beginning of the Page Buffer

E Moves Dot to the End of the Page Buffer (see also /J and /A below)

4.3.2.2 Jump and Advance -

nJ Jumps Dot forward past n characters

-nJ Moves Dot backward past n characters

nA Advances Dot forward past n ends-of-lines to the beginning of the succeeding line

-nA Moves Dot backwards across n ends-of-lines and positions Dot immediately after n+1 ends-of-lines, i.e., at the beginning of the -n line.

0J or 0A Moves Dot to the beginning of the current line

@J or @A Moves Dot to the Marked location

/J or /A Moves Dot to the end of the Page Buffer (see also E above)

Notice that while n moves Dot n characters in the Jump command, its role becomes that of a line counter in the Advance command. However, because 0, @, and / are absolute, their use with these commands overrides line/character distinctions. That is, Jump and Advance perform identical functions if both have either 0, @ or / for an argument.

4.3.2.3 Mark - The M command marks ("remembers") the current position of Dot for later reference in a command using the argument @. Note that only one position at a time can be in a marked state. Mark is also affected by the execution of those commands which alter the contents of the Page Buffer:

C D H I K N R X

4.3.3 Search Commands

4.3.3.1 Get - The basic search command nG starts at Dot and Gets the nth occurrence of the specified text in the Page Buffer. If no argument is present, it is assumed to be 1. When you type the command, followed by the RETURN key, ED-11 goes into Text Mode. The character string to be searched for must now be typed. (ED-11 will accept a search object of up to 42 characters.) Typing the LINE FEED key terminates Text Mode and initiates the search.

This command sets Dot to the position immediately following the found character string, and a 0L listing is performed by ED-11. If a carriage return, line feed, or form feed is specified as part of the search object, the automatic 0L displays only a portion of text -- the part defined as the last line. Where any of these characters is the last character of the search object, the 0L of course yields no printout at all.

If the search is unsuccessful, Dot is at the end of the Page Buffer and a ? is printed out. The Editor then returns to Command Mode.

Examples:

```
1.          Text                     Command        Printout

       MOV @RMAX,@R5                  2G↵             BEQ CK
       ADD #6,(R5)+                   CK
       CLR $CK3
       TST R2
       BEQ CKCR

       Dot was here.                               Dot is now here.

2.     CMPB    ICHAR,#RUBOUT          G↵              BR
       BEQ     SITE                   TE↵
       BR      PUT                    BR↓

      Dot                                           Dot
```

4.3.3.2 wHole - A second search command, H, starts at Dot and looks through the wHole text file for the next occurrence of the character string you have specified in Text Mode. It combines a Get and a Next such that if the search is not successful in the Page Buffer, the contents of the buffer are punched on tape, the buffer contents are deleted, and a new page is read in, where the search is continued. This continues until the search object is found or until the complete source text has been searched. In either case, Mark is at the beginning of the Page Buffer.

If the search object is found, Dot is located immediately following it, and a 0L is performed by ED-11. As in the Get command, if the search is not successful Dot is at the end of the buffer and a ? appears on the teleprinter. Upon completion of the command, the Editor will be in Command Mode. No argument is allowed. Note that an H command specifying a nonexistent search object can be used to close out an edit, i.e., copy all remaining text from the input tape to the output tape.

4.3.4 Commands to Modify the Text

4.3.4.1 Insert - The Insert command (I) allows text to be inserted at Dot. After I is typed (followed by the typing of the RETURN key), the Editor goes into Text Mode to receive text to be inserted. Up to 80 characters per line are acceptable. Execution of the command occurs when the LINE FEED key (which does not Insert a line feed character unless it is the first key typed in Text Mode) is typed terminating Text Mode. At this point, Dot is located in the position immediately following the last inserted text character. If the Marked location was anywhere after the text to be Inserted, Dot becomes the new Marked location.

During an insert, it sometimes happens that the user accidentally types CTRL/P rather than SHIFT/P (for @), thus deleting the entire insert (see Section 4.4.1). To minimize the effect of such a mistake, the insert may be terminated every few lines and then continued with a new Insert command.

As with the Read command, an attempt to overflow the Page Buffer causes a ? to be printed out followed by an * on the next line indicating that a command may be typed. All or part of the last line typed may be lost. All previously typed lines are inserted. Examples:

```
             Text                      Command              Effect

     1.  MOV #8.,EKOT                   I↵               MOV #8.,EKOCNT
                    Dot                 CN               Dot

     2.  Inserting a carriage return (and automatic line feed):

         CLR R1CLR R2                   I↵               CLR R1
                                        ↵                CLR R2
              Dot                       ↓

     3.  Inserting a single line feed:

                                        I↵
         LOOK WHAT HAPPENS HERE         ↓                LOOK WHAT
                                        ↓                HAPPENS HERE
         Dot
                                                           Dot
```

4.3.4.2 Delete and Kill - These commands are closely related to each other; they both erase specified text from the Page Buffer. The Delete command (D) differs from the Kill command (K) only in that the former accepts an argument, n, that counts characters to be removed, while the latter accepts an argument, n, that counts lines to be removed. 0, @, and / are also allowed as arguments. After execution of these commands, Dot becomes the Marked location.

nD Deletes the following n characters

-nD Deletes the previous n characters

nK Kills the character string beginning at Dot and ending at the nth end-of-line

-nK Kills the character string beginning with the first character following the (n+1)th previous end-of-line and ending at Dot

0D or 0K Removes the current line up to Dot

@D or @K Removes the character string bounded by Dot and Mark

/D or /K Removes the character string beginning at Dot and ending with the last character in the Page Buffer

```
        Text                     Command        Effect

1.  ;CHECK THE MOZXDE              -2D      ;CHECK THE MODE

              Dot                            Dot

2.  ;IS IT A TAB OR                2K       ;IS IT A TAB
    ;IS IT A CR

              Dot                            Dot
```

4.3.4.3 Change and exchange - The Change (C) and eXchange (X) commands can be thought of as two-phase commands combining, respectively, an Insert followed by a Delete, and an Insert followed by a Kill. After the Change or eXchange command is typed, ED-11 goes into Text Mode to receive the text to be inserted. If n is used as the argument, it is then interpreted as in the Delete (character-oriented) or Kill (line-oriented), and accordingly removes the indicated text. 0, @, and / are also allowed as arguments.

```
nC          Changes the following
xxxx        n characters

xxxx

-nC         Changes the previous
xxx         n characters
nX          eXchanges the character
xxxx        string beginning at Dot and
xxxx        ending at the nth end-of-line

-nX         eXchanges the character
xxx         string beginning with the first character
            following the (n+1)th previous end-of-line and
            ending at Dot
```

```
OC or 0X        Replaces the current line up to Dot
xxxx  xxxx
xxxx  xxxx

@C  or  @X      Replaces the character string bounded by Dot
xxx     xxx     and the Marked location
xxx     xxx

/C  or  /X      Replaces the character string beginning at Dot
xxx     xxx     and ending with the last character in the Page
                Buffer.
```

Again, the use of absolute arguments 0, and @, and / overrides the line/character distinctions that n and -n produce in these commands.

If the Insert portion of a Change or eXchange is terminated because of attempting to overflow the Page Buffer, data from the latest line may have been lost, and text removal does not occur. Such buffer overflow might be avoided by separately executing a Delete or Kill followed by an Insert, rather than a Change or eXchange, which does an Insert followed by a Delete or Kill. Examples:

```
             Text                 Command          Effect

     ;A LINE FEED IS HERE         -9C↵           ;  A TAB IS HERE
                                   TAB↓
     ;THIS                         2X↵           ;THIS
     ;IS ON       Dot             PAPER          ;IS ON
     ;FOUR                                       ;PAPER
     ;LINES

     Dot                                         Dot
```

4.4 OPERATING PROCEDURES

4.4.1 Error Corrections

During the course of editing a page of the program, it may become necessary to correct mistakes in the commands themselves. There are four special commands which do this:

1. Typing the RUBOUT key removes the preceding typed character, if it is on the current line. Successive RUBOUTs remove preceding characters on the line (including the SPACE), one character for each RUBOUT typed.

2. The CTRL/U combination (holding down the CTRL key and typing U) removes all the characters in the current line.

3. CTRL/P cancels the current command in its entirety. This includes all the current command text just typed, if ED-11 was in Text Mode. Do not use another CTRL/P before typing a line terminator as this will cause an ED-11 restart (see 4. below). If CTRL/P is typed while a found search object of a Get or wHole is being printed out, the normal position of Dot (just after the specified search object) is not affected.

   CTRL/P should not be used while a punch operation is in progress as it is not possible to know exactly how much data will be output.

4. Two CTRL/P's not interrupted by a typed line terminator restart ED-11, initiating the dialogue described in Section 4.4.2.

After removing the incorrect command data, the user can directly type in the desired input.

4.4.2 Starting

The Editor is loaded by the Absolute Loader (see Chapter 6, Section 6.2.2) and starts automatically. Once the Editor has been loaded, the following sequence occurs:

ED-11 Prints	User Types	
*I	L↵	(if the low-speed Reader is to be used for source input)
	H↵	(if the high-speed Reader is to be used for source input)
*O	L↵	(if the low-speed Punch is to be used for edited output)
	H↵	(if the high-speed Punch is to be used for edited output)

If all text is to be entered from the keyboard (i.e., via the Insert command), either L or H may be specified for Input.

If the output device is the high-speed punch (HSP), the Editor enters Command Mode to accept input. Otherwise, the sequence continues with:

LSP OFF ↵ (when low-speed Punch (LSP) is off)

Upon input of ↵ from the keyboard, the Editor enters Command Mode and is ready to accept input.

4.4.3 Restarting

To restart ED-11, type CTRL/P twice. This initiates the normal starting dialogue described in Section 4.4.2. If the Low-speed Reader (LSR) is in operation it must first be turned off. The text to be edited should be loaded (or reloaded) at this time.

4.4.4 Creating a Paper Tape

Input commands assume that text is to be read from a paper tape by the low-speed reader or high-speed reader. However, the five commands that go into Text Mode enable the user to input from the keyboard. The Insert command, in particular (Section 4.3.4.1) can be useful for entering large quantities of text not on paper tape. The Page Buffer can thus be filled from the keyboard, and a paper tape actually created by using a command to punch out the buffer contents.

4.4.5 Editing Example

The following example consists of three parts:

1. The marked up source program listing indicating the desired changes.

2. The ED-11 commands to implement those changes (with comments on the editing procedure).

   NOTE

   Typing the RETURN key terminates Command Mode in all cases. In commands which then go into Text Mode, typing the LINE FEED key (symbolized as ↓) produces the terminator.

3. The edited text.

Part I Original Source for Edit

```
        ;COMMON INPUT ROUTINE FOR USE BY NON FILE DEVICES

$INPUT: ADC     ICHAR,(R5)+       ;UPDATE CKSUM
        CLR     -(LS)             ;CLEAR DONE
        MOV     (R5)+,RMAX        ;GET ADR MAX
        MOV     (R5)+,MODADR      ;GET ADR MODE
                                  ;R5 NOW POINTS TO POINTER

$CKMODE:BITB    @MODADB,#ASCII    ;IS THIS ASCII
        BNE     CKBIN             ;NO---TRY BINARY

$CKNUL: TSTB    ICHAR             ;ASCII---IS CHAR A NULL
        BEQ     CK                ;YES--NO GO

                                  ;LOOK AT MODE TO SEE IF
$CKPAR: BITB    @MODADR,#PARBIT   ;SUPPOSED TO CHECK PARITY?
        BNE     PAROK             ;NO
        MOVB    ICHAR,OCHAR       ;YES---CK IT
        JSR     R7,PARGEN
        SUB     ICHAR,OCHAR       ;
        BEQ     PAROK             ;OK?
        BIS     #PARERR,@MODADR   ;NO---SET ERR BIT
PAROK:  CLR     OCHAR
        BIC     #177200,ICHAR     ;STRIP PARITY
        CMPB    @10(RADD),#KBD    ;IS THIS KBD INPUT
        BNE     OKO               ;NO
        TSTB    EKOCNT            ;YES---DONE EKO OF LAST?
        BEQ     $OK               ;YES
        CLR     ICHAR             ;NO---DROP NEW CHAR
$JP2CK: JMP     CK^                                         DUN

        ;WHAT IS THE CHAR
$OK:    CMPB    ICHAR,#CTRLC      ;IS IT A ^C
        BNE     ~~OKUPP~~             ;NO                   OKØ
        MOV     #UPC,OCHAR        ;YES--ECHO ^C
        INC     RDUN
        MOV     #ABRTAD,20(R6)    ;DIDDLE RETURN ADR
        BB      PLUS1
```

```
~~CKUPP:  CMPB     ICHAR,#CTRLP       ;IS IT A ^P~~
          BNE      CK1                ;NO
          TST      RESTAD             ;YES--DID HE SET UP
          BFQ      OKO                ;A RESTART ADR?
          MOV      RESTAD.20(R6)      ;YES---XFB THERE
          CLR      ICHAR
          INC      RDUN
          MOV      #UPP,OCHAR
~~        BB       PLUS1~~

                                      ;THIS IS NOT KBD INPUT
~~OKO:    BITB     @MODADR,#FORMAT    ;IS THIS ASCII FORMATTED?~~   ; FORMATTED AND UNFORMATTED
~~        BEQ      CKINP              ;YES...DO CHAR CONV~~
~~                                    ;NO   IT IS UNFORMATTED~~     ; ASCII ARE HANDLED THE SAME

          CMPB     ICHAR,#RUBOUT      ;IS THIS A RUBOUT
          BEQ      CK                 ;YES---IGNORE IT
          BB       PUT                ;NO---

~~CKINP:  CMPB     ICHAR,#RUBOUT      ;YES   IS CHAR A RUBOUT?~~
          BNE      CKUPU              ;NO
~~        CLR      ICHAR              ;YES~~
~~        TST      2(R5)              ;BO 0?~~
          BEQ      CK                 ;YES---FORGET IT
          MOVB     #BSLASH,OCHAR      ;ECHO A \
          DEC      (R5)+              ;POINTER=POINTER-1
          DEC      @R5                ;BC=BC-1
          BB       EKO                ;EKO

CKUPU:    CMPB     ICHAR,#CTRLU       ;IS IT A ^U?
          BNE      CKTAB              ;NO
          MOV      #UPU,OCHAR         ;YES---ECHO ^U
          CLR      ICHAR
          MOV      @RMAX,@R5          ;POINTER=BUFADR+6
          ADD      #6,(R5)+
          CLR      @R5                ;BC=0
          BB       EKO                ;ECHO

CKTAB:    CMPB     ICHAR,#HTAB        ;IS IT A TAB
          BNE      CKCR               ;NO
          MOV      #BLNKS,OCHAR       ;YES---ECHO BLANKS
          MOV      TABCNT,EKOCNT      ;SET UP COUNTER
          BB       PUT                ;

CKCR:     CMPB     ICHAR,#CR          ;IS IT A CR?
          BNE      $CK3               ;NO
          MOV      #CRLF,OCHAR        ;YES---ECHO CRLF
          INC      RDUN
          BB       PLUS1              ;
                                                                 ALT
$CK3:     CMPB     ICHAR,#033          ^                         ; IS CHAR AN ALTMODE?
          BEQ      $ALT
          CMPB     ICHAR,#175
          BEQ      $ALT
          CMPB     ICHAR,#176
          BNE      CKUP                          EX
~~$ALT:   MOV      #DBL,OCHAR~~
          MOV      #175,ICHAR
^ ~~      INC      RDUN~~                            $ALT:
~~        BB       PUT~~
```

```
CKLF:   CMPB    ICHAR,#LF
        BNE     CKFF
        INC     RDUN
        BB      PUT

CKFF:   MOV     ICHAR,OCHAR
        CMPB    ICHAR,#FF
        BNE     PUT
        MOV     #8.,EKOCNT
        MOV     #LFLF,OCHAR
        BB      PUT
```

Part II: Editing Session

Assume that ED-11 has been started, is in Command Mode, and the tape is in the reader. Underlined matter indicates ED-11 output.

```
*R                           ;Reads in a page of text

*H                           ;Searches entire program for 2CK: -
2CK:↓                        ;when found ED-11 performs a 0L
$JP2CK:

*G                           ;Searches current page for next CK -
CK                           ;when found ED-11 performs a 0L
$JP2CK  JMP     CK

*I                           ;Inserts DUN following CK
DUN↓

*G                           ;Searches for next CKUPP -
CKUPP↓                       ;when found ED-11 performs a 0L
          BNE     CKUPP

*-5C                         OK0 replaces last 5 characters (CKUPP)
OK0

*6A                          ;Dot is moved 6 lines ahead (including
                             ;a blank line)

*9K                          ;9 lines are killed starting with CKUPP:

*L                           ;Next line is listed - Dot is not moved
                             ;THIS IS NOT KBD INPUT

*I                           ;Blank line is inserted
↵
↓
*A                           ;Dot is moved 1 line ahead to point to
                             ;character O of OK0:

*4X                          ;Following comments replace the next 4
                             ;lines
                             ;FORMATTED AND UNFORMATTED
                             ;ASCII ARE HANDLED THE SAME↓

*G                           ;Searches for next CKINP: -
CKINP:↓                      0L printout occurs when found
CKINP:
```

```
*0J                      ;Dot is moved to the beginning of the
                         ;current line.

*/K                      ;The rest of the page is killed (3 lines)

*N                       ;Current page is punched out on paper tape -
                         ;a new page is read in

*L                       ;The next line is listed - Dot is not moved
     TST 2(R5)    ;BC=0?

*15K                     ;15 lines are killed starting with TST

*2L                      ;1 blank line and 1 line of text
                         ;are listed - Dot is not moved

CKTAB: CMPB ICHAR,#HTAB  ;IS IT A TAB

*2G                      ;Searches for 2nd occurrence of $CK3 -
$CK3↓                    ;0L printout verifies it is found
$CK3

*-C                      ;ALT replaces preceding character
ALT↓

*V                       ;Lists entire current line to verify
$CKALT: CMPB ICHAR,#033  ;the above-C result

*G                       ;Searches for the 033 to position Dot
033↓                     ;for next command -- 0L occurs
$CKALT: CMPB ICHAR,#033

*I                       ;The following text is inserted in the
                         ;comment field
                              ;IS CHAR AN ALTMODE?

*G                       ;Searches for next CKLF -- 0L occurs
CKLF↓
          BNE CKLF

*-2C                     ;EX replaces the preceding two characters
EX                       ;(LF)

*2J                      ;Jumps Dot past the carriage return and
                         ;line feed characters

*K                       ;Kills next line (starting with $ALT:)

*I                       ;Inserts $ALT: at beginning of the fol-
$ALT:↓                   ;lowing line

*A                       ;Advances Dot past 1 line feed to the
                         ;beginning of the next line

*M                       ;Marks the position of Dot

*B                       ;Moves Dot to the beginning of the cur-
                         ;rent page
```

```
*@P                     ;Punches out the lines from Dot to the
                        ;position just marked - Dot not moved

*@A                     ;Moves Dot from the beginning of the
                        ;page to the marked position

*2K                     ;Kills the next 2 lines
*
```

PART III Edited Source

```
        ;COMMON INPUT ROUTINE FOR USE BY NON FILE DEVICES

$INPUT: ADD     ICHAR,(R5)+        ;UPDATE CKSUM
        CLR     -(LS)              ;CLEAR DONE
        MOV     (R5)+,RMAX         ;GET ADR MAX
        MOV     (R5)+,MODADR       ;GET ADR MODE
                                   ;R5 NOW POINTS TO POINTER

$CKMODE:BITB    @MODADB,#ASCII     ;IS THIS ASCII
        BNE     CKBIN              ;NO---TRY BINARY

$CKNUL: TSTB    ICHAR              ;ASCII---IS CHAR A NULL
        BEQ     CK                 ;YES--NO GO

                                   ;LOOK AT MODE TO SEE IF
$CKPAR: BITB    @MODADR,#PARBIT    ;SUPPOSED TO CHECK PARITY?
        BNE     PAROK              ;NO
        MOVB    ICHAR,OCHAR        ;YES---CK IT
        JSR     R7,PARGEN
        SUB     ICHAR,OCHAR        ;
        BEQ     PAROK              ;OK?
        BIS     #PARERR,@MODADR    ;NO---SET ERR BIT
PAROK:  CLR     OCHAR
        BIC     #177200,ICHAR      ;STRIP PARITY
        CMPB    @10(RADD),#KBD     ;IS THIS KBD INPUT
        BNE     OKO                ;NO
        TSTB    EKOCNT             ;YES---DONE EKO OF LAST?
        BEQ     $OK                ;YES
        CLR     ICHAR              ;NO---DROP NEW CHAR
$JP2CK: JMP     CKDUN

        ;WHAT IS THE CHAR
$OK:    CMPB    ICHAR,#CTRLC       ;IS IT A ^C
        BNE     OKO                ;NO
        MOV     #UPC,OCHAR         ;YES--ECHO ^C
        INC     RDUN
        MOV     #ABRTAD,20(R6)     ;DIDDLE RETURN ADR
        BB      PLUS1

                                   ;THIS IS NOT KBD INPUT
                                   ;FORMATTED AND UNFORMATTED
                                   ;ASCII ARE HANDLED THE SAME
        CMPB    ICHAR,#RUBOUT      ;IS THIS A RUBOUT
        BEQ     CK                 ;YES---IGNORE IT
        BB      PUT                ;NO---
CKTAB:  CMPB    ICHAR,#HTAB        ;IS IT A TAB
        BNE     CKCR               ;NO
        MOV     #BLNKS,OCHAR       ;YES---ECHO BLANKS
        MOV     TABCNT,EKOCNT      ;SET UP COUNTER
        BB      PUT                ;
```

```
CKCR:    CMPB     ICHAR,#CR          ;IS IT A CR?
         BNE      $CK3               ;NO
         MOV      #CRLF,OCHAR        ;YES---ECHO CRLF
         INC      RDUN
         BB       PLUS1              ;

$CKALT:  CMPB     ICHAR,#033         ;IS CHAR AN ALTMODE?
         BEQ      $ALT
         CMPB     ICHAR,#175
         BEQ      $ALT
         CMPB     ICHAR,#176
         BNE      CKEX
$ALT:    MOV      #175,ICHAR
CKLF:    CMPB     ICHAR,#LF
         BNE      CKFF
         INC      RDUN
         BB       PUT

CKFF:    MOV      ICHAR,OCHAR
         CMBP     ICHAR,#FF
         BNE      PUT
         MOV      #8,EKOCNT
         MOV      #LFLF,OCHAR
         BB       PUT
```

4.5 SOFTWARE ERROR HALTS

ED-11 loads all unused trap vectors with the code

```
          .WORD       .+2,HALT
```

so that if the trap does occur, the processor halts in the second word of the vector. The address of the halt, displayed in the console address register, therefore indicates the cause of the halt. In addition to the halts which may occur in the vectors, the standard IOX error halt at location 40 may occur (see Chapter 7).

Address of HALT	Meaning
12	Reserved instruction executed
16	Trace trap occurred
26	Power fail trap
32	EMT executed
36	TRAP executed
40	IOX detected error

CHAPTER 5

DEBUGGING OBJECT PROGRAMS ON-LINE

5.1 INTRODUCTION

ODT-11 (On-line Debugging Technique for the PDP-11) is a system program which aids in debugging assembled object programs. From the Teletype keyboard you interact with ODT and the object program to:

- print the contents of any location for examination or alteration
- run all or part of an object program using the breakpoint feature
- search the object program for specific bit patterns
- search the object program for words which reference a specific word
- calculate offsets for relative addresses

During a debugging session you should have at the terminal the assembly listing of the program to be debugged. Minor corrections to the program may be made on-line during the debugging session. The program may then be run under control of ODT to verify any change made. Major corrections, however, such as a missing subroutine, should be noted on the assembly listing and incorporated in a subsequent updated program assembly.

A binary tape of the debugged program can be obtained by use of the DUMPAB program (see Chapter 6, section 6.3).

5.1.1 ODT-11 and ODT-11X

There are two versions of ODT included in the PDP-11 Paper Tape Software System: a standard version, ODT-11, and an extended version, ODT-11X.[1] Both versions are independent, self-contained programs. ODT-11X has all the features of ODT-11, plus some additional features. Each version is supplied on two separate paper tapes: a source tape and an absolute binary tape. The purpose of the tapes, and loading and starting procedures are explained in a later section of this chapter.

ODT-11 is completely described in section 5.2, and the additional features of ODT-11X are covered in section 5.3. In all sections of this chapter, except where specifically stated, reference to ODT

[1] Only ODT-11X is available for the LSI-11 or the PDP-11/03.

applies to both versions. Concluding sections discuss ODT's internal operations -- how it effects breakpoints, how it uses the "trace trap" and the T-bit, and other useful data.

The following discussion assumes that the reader is familiar with the PDP-11 introduction formats and the PAL-11A Assembly Language as described in Chapter 3.

5.1.2 ODT's Command Syntax

ODT's commands are composed of the following characters and symbols. They are often used in combination with the address upon which the operation is to occur, and are offered here for familiarization prior to their thorough coverage which follows. Unless indicated otherwise, n below represents an octal address.

n/ open the word at location n

/ reopen last opened location

n\ (SHIFT/L) open the byte at location n (ODT-11X only)

\ reopen the last opened byte (ODT-11X only)

↓ (LINE FEED key) open next sequential location

↑[1] open previous location

RETURN close open location and accept the next command

←[2] take contents of opened location, index by contents of PC, and open that location

@ take contents of opened location as absolute address and open that location (ODT-11X only)

> take contents of opened location as relative branch instruction and open referenced location (ODT-11X only)

< return to sequence prior to last @, >, or _ command and open succeeding location (ODT-11X only)

$n/ open general register n (0-7)

; separates commands from command arguments (used with alphabetic commands below)

;B remove Breakpoint(s) (see description of each ODT version for particulars)

n;B set Breakpoint at location n

[1] The circumflex appears on some keyboards and printers in place of the up-arrow.

[2] The underline appears on some keyboards and printers in place of the back-arrow.

n;rB set Breakpoint r at location n (ODT-11X only)

;rB remove r(th) Breakpoint (ODT-11X only)

n;E search for instructions that reference Effective address n

n;W search for Words with bit patterns which match n

;nS enable Single-instruction mode (n can have any value and is not significant); disable breakpoints

;S disable Single-instruction mode

n;G Go to location n and start program run

;P Proceed with program execution from breakpoint; stop when next breakpoint is encountered or at end of program

In Single-instruction mode only (ODT-11X), Proceed to execute next instruction only

n;P Proceed with program execution from breakpoint; stop after encountering the breakpoint n times.

In single-instruction mode only (ODT-11X), Proceed to execute next n instructions.

n/(word)n;O calculate Offset from location n to location m

$B/ open Breakpoint status word (ODT-11)
open BREAKPOINT 0 STATUS WORD (ODT-11X)

$M/ open search Mask

$S/ open location containing user program's Status register

$P/ open location containing ODT's Priority level

With ODT-11, location references must be to even numbered 16-bit words. With ODT-11X, location references may be to 16-bit words or 8-bit bytes.

The semicolon in the above commands is ignored by ODT-11, but is used for the sake of consistency, since similar commands to ODT-11X require it.

5.2 COMMANDS AND FUNCTIONS

When ODT is started as explained in section 5.6, it indicates its readiness to accept commands by printing an asterisk (*) on the left margin of the terminal paper. In response to the asterisk, you can issue most commands; for example, you can examine and, if desired, change a word, run the object program in its entirety or in segments, or even search core for certain words or references to certain words. The discussion below first explains some elementary features, and then covers the more sophisticated features.

All commands to ODT are issued using the characters and symbols shown above in Section 5.1.2.

5.2.1 Opening, Changing, and Closing Locations

An open location is one whose contents ODT has printed for examination, and whose contents are available for change. A closed location is one whose contents are no longer available for change. Any even-numbered location may be opened using ODT-11.

The contents of an open location can be changed by typing the new contents followed by a single character command, which requires no argument (i.e. ↓ ↑ RETURN ← @ > <). Any command typed to open a location when another location is already open causes the currently open location to be closed.

5.2.1.1 The Slash (/) - One way to open a location is to type its address followed by a slash:

```
     *1000/012746
```

Location 1000 is open for examination and is available for change. Note that in all examples ODT's printout is underlined; your typed input is not.

Should you not wish to change the contents of an open location, merely type the RETURN key and the location will be closed; ODT prints another asterisk and waits for another command. However, should you wish to change the word, simply type the new contents before giving a command to close the location.

```
     *1000/012746    012345
     *
```

In the example above, location 1000 now contains 012345 and is closed since the RETURN key was typed after entering the new contents, as indicated by ODT's second asterisk.

Used alone, the slash reopens the last location opened:

```
     *1000/012345   2340
     */002340
```

As shown in the example above, an open location can be closed by typing the RETURN key. In this case, ODT changed the contents of location 1000 to 002340 and then closed the location before printing the *. A single slash then directed ODT to reopen the last location opened. This allowed us to verify that the word 002340 was correctly stored in location 1000. (ODT supplies the leading zeroes if not given.)

Note again that opening a location while another is currently open automatically closes the currently open location before opening the new location.

5.2.1.2 The LINE FEED Key - If the LINE FEED key is typed when a location is open, ODT closes the open location and opens the next sequential location:

```
     *1000/002340 ↓        ( ↓ denotes typing the LINE FEED key)
     001002/012740
```

In this example, the LINE FEED key instructed ODT to print the address of the next location along with its contents and to wait for further instructions. After the above operation, location 1000 is closed and 1002 is open. The open location may be modified by typing the new contents.

5.2.1.3 The Up-Arrow(↑) - The up-arrow (or curcumflex) symbol is effected by typing the SHOFT and N key combination. If the up-arrow is typed when a location is open, ODT closes the open location and opens the previous location (as shown by continuing from the example above):

```
001002/012740    ↑ (↑ is printed by typing SHOFT and N)
001000/002340
```

Now location 1002 is closed and 1000 is open. The open location may be modified by typing the new contents.

5.2.1.4 The Back-Arrow(←) - The back-arrow (or underline) symbol is effected by typing the SHIFT and O key combination. If the back-arrow is typed to an open location, ODT interprets the contents of the currently open location as an address indexed by the Program Counter (PC) and opens the location so addressed:

```
*1006/000006    ←    ( ← is printed by typing SHIFT and O)
001016/100405
```

Notice in this example that the open location(1006) was indexed by the PC as if it were the operand of an instruction with address mode 67 as explained in Chapter 3.

A modification to the opened location can be made before a ↓ , ↑ , or ← is typed. Also, the new contents of the location will be used for address calculations using the _ command. Example:

```
*100/000222   4↓       (modify to 4 and open next location)
000102/000111  6↑      (modify to 6 and open previous location)
000100/000004   100←   (change to 100 and open location indexed
000202/(contents)       by PC)
```

5.2.1.5 Accessing General Registers 0-7 - The program's general registers 0-7 can be opened using the following command format:

```
*$n/
```

where n is the integer representing the desired register (in the range 0 through 7). When opened, these registers can be examined or changed by typing in new data as with any addressable location. For example:

```
*$0/000033          (R0 was examined and closed)
*
```

and

```
*$4/000474 464      (R4 was opened, changed, and closed)
*
```

The example above can be verified by typing a slash in response to ODT's asterisk:

```
*/000464
```

The ↓ , ↑ , ← , or @ commands may be used when a register is open (the @ is an ODT-11X command).

5.2.1.6 Accessing Internal Registers - The program's Status Register contains the condition codes of the most recent operational results and the interrupt priority level of the object program. It is opened using the following command:

```
*$S/000311
```

where $S represents the address of the Status Register. In response to $S/ in the example above, ODT printed the 16-bit (of which only the low-order 8 bits are meaningful): Bits 0-3 indicate whether a carry, overflow, zero, or negative (in that order) has resulted, and bits 5-7 indicate the interrupt priority level (in the range 0-7) of the object program.

The $ is used to open certain other internal locations:

$B internal breakpoint status word (see section 5.2.2.2)

$M mask location for specifying which bits are to be examined during a bit pattern search (see section 5.2.4)

$P location defining the operating priority of ODT (see section 5.2.6)

$S location containing the condition codes (bits 0-3) and interrupt priority level (bits 5-7)

5.2.2 Breakpoints

The breakpoint feature facilitates monitoring the progress of program execution. A breakpoint may be set at any instruction which is not referenced by the program for data. When a breakpoint is set, ODT replaces the contents of the breakpoint location with a trap instruction. Thus, when the program is executed and the breakpoint is encountered, program execution is suspended, the original contents of the breakpoint location are restored, and ODT regains control.

5.2.2.1 Setting the Breakpoint(n;B) - ODT-11 provides only one breakpoint; ODT-11X provides eight. Breakpoint(s) may be changed at any time. A breakpoint is set by typing the address of the desired location of the breakpoint followed by ;B. For example:

```
*1020;B
*
```

sets a breakpoint at location 1020. The breakpoint above is changed to location 1120 as shown below.

```
*1020;B
*1120;B
*
```

Breakpoints should not be set at locations referenced by the program for data, nor at an IOT, EMT, or TRAP instruction. This restriction is explained in section 5.5.2.

The breakpoint is removed by typing ;B without an argument, as shown below.

```
*1120;B         (sets breakpoint at location 1120)
*;B             (removes breakpoint)
*
```

5.2.2.2 Locating the Breakpoint($B) - The command $B/ causes ODT-11 to print the address of the breakpoint (see also section 5.3.3 on $B in ODT-11X):

```
*$B/001120
```

The breakpoint was set at location 1120. $B represents the address containing ODT-11's breakpoint location. Typing the RETURN key in the example above leaves the breakpoint at location 1120 and returns control to ODT-11. The breakpoint could be changed to a different location:

```
*$B/001120   1114
*$B/001114
*
```

The breakpoint was found in location 1120, changed to location 1114, and the change was verified.

If no breakpoint is set, $B contains an address internal to ODT-11.

5.2.3 Running the Program(n;G and n;P)

Program execution is under control of ODT. There are two commands for running the program: n;G and n;P. The n;G command is used to start execution (GO) and n;P to continue (Proceed) execution after halting at a breakpoint. For example:

```
*1000;G
```

starts execution at location 1000. The program runs until it encounters a breakpoint or until program completion. If the program enters an infinite loop, it must be restarted or reentered as explained in section 5.6.2.

When a breakpoint is encountered, execution stops and ODT-11 prints B; followed by the address of the breakpoint. Desired locations can then be examined for expected data. For example:

```
*1010;B             (breakpoint is set at location 1010)
*1000;G             (execution started at location 1000)
B;001010            (execution stopped at location 1010)
*
```

To continue program execution from the breakpoint, type ;P in response to ODT-11's last *.

When a breakpoint is set in a loop, it may be desirable to allow the program to execute a certain number of times through the loop before recognizing the breakpoint. This may be done by typing the n;P command and specifying the number of times the breakpoint is to be encountered before program execution is suspended (on the n(th) encounter). (See section 5.3.3 for ODT-11X interpretation of this command when more than one breakpoint is set in a loop.)

Example:

```
B;001010                (execution halted at breakpoint)
*1250;B                 (set breakpoint at location 1250)
*4;P                    (continue execution.  loop through
B;001250                 breakpoint 3 times and halt on the
*                        4(th) occurrence of the breakpoint)
```

The breakpoint repeat count can be inspected by typing $B/ followed by LINE FEED. The repeat count is then printed. This also provides an alternative way of specifying the count. Since the location is open, its contents can be modified in the usual manner by typing new contents followed by the RETURN key.

```
*$B/001114  ↓           (address of breakpoint is 1114)
nnnnnn/000003   6       (repeat count was 3, changed to 6)
*
```

Breakpoints are inserted when performing an n;G or n;P command. Upon execution of the n;G or n;P command, the general registers 0-6 are set to the values in the locations specified as $0-$6 and the processor status register is set to the value in the location specified as $S.

5.2.4 Searches

With ODT you can search all or any specyfied portion of core memory for any specific bit pattern or for references to a specific location.

The location represented by $M is used to specify the mask of the search. The next two sequential locations contain the lower and upper limits of the search. Bits set to 1 in the mask are examined during the search; other bits are ignored. For example,

```
*$M/000000   177400 ↓          (↓ denotes typing LINE FEED)
nnnnnn/000000  1000  ↓         (starting address of search)
nnnnnn/000000  1040            (last address in search)
*
```

where nnnnnn represents some location in ODT. This location varies and is meaningful only for reference purposes. Note that in the first line above, the slash was used to open $M which now contains 177400, and that the LINE FEEDs opened the next two sequential locations which now contain the lower and upper limits of the search.

5.2.4.1 Word Search(n;W) - Before initiating a word search, the mask and search limits must be specified as explained above. Then the search object and the initiating command are given using the n;W command where n is the search object. When a match is found, the address of the unmasked matching word is printed. For example:

```
*$M/000000   177400 ↓      (test high order eight bits)
nnnnnn/000000  1000  ↓
nnnnnn/000000  1040
*400;W                     (initiating word search)
001010/000770
001034/000404
*
```

In the search process, the word currently being examined and the search object are exclusive ORed (XORed), and the result is ANDed to the mask. If this result is zero, a match has been found, and is reported at the terminal. Note that if the mask is zero, all locations within the limits are printed.

5.2.4.2 Effective Address Search(n;E) - ODT enables you to search for words which address a specified location. After specifying the search limits (section 5.2.4), type n;E (where n is the effective address) to initiate the search.

Words which are either an absolute address (argument n itself), a relative address offset, or a relative branch to the effective address are printed after their addresses. For example:

```
*$M/177400  ↓
nnnnnn/001000   1010 ↓
nnnnnn/001040   1060
*1034;E                     (initiating search)
001016/001006               (relative branch)
001054/002767               (relative branch)
*1020;E                     (initiating a new searhc)
001022/177774               (relative address offset)
001030/001020               (absolute address)
*
```

Particular attention should be given to the reported references to the effective address because a word may have the specified bit pattern of an effective address without actually being so used. ODT will report these as well.

5.2.5 Calculating Offsets(n;O)

Relative addressing and branching use an offset - the number of wrods or bytes forward or backward from the current location of the effective address. During the debugging session it may be necessary to change a relative address or branch reference by replacing one instruction offset with another. ODT calculates the offsets in response to the n;O command.

The command n;O causes ODT to print the 16-bit and 8-bit offsets from the currently open location to address n. In ODT-11, the 8-bit offset is printed as a 16-bit word. For example:

```
*346/000034   414;O    000044   000022    22
*/000022
*20/000046    200;O    000156   000067    67
*20/000067
```

In the first example, location 346 is opened and the offsets from that location to location 414 are calculated and printed. The contents of location 346 are then changed to 22 and verified on the next line. The 16-bit offset is printed followed by the 8-bit offset. In the example above, 000156 is the 16-bit offset and 000067 is the 8-bit offset.

The 8-bit offset is printed only if the 16-bit offset is even, as in the case above. With ODT-11 only, the user must determine whether the 8-bit offset is out of the range 177600 to 000177 (-128 decimal to 127 decimal). The offset of a relative branch is calculated and modified as follows:

```
*1034/103421   1034;O   177776   177777   103777
*
```

Note that the modified low-order byte 377 must be combined with the unmodified high-order byte. Location 1034 was still open after the calculation, thus typing 103777 changed its contents; the location was then closed.

5.2.6 ODT'S Priority Level($P)

$P represents a location in ODT that contains the priority level at which ODT operates. If $P contains the value 377, ODT operates at the priority level of the processor at the time ODT is entered. Otherwise $P may contain a value between 0 and 7 corresponding to the fixed priority at which ODT operates.

To set ODT to the desired priority level, open $P. ODT prints the present contents, which may then be changed:

```
*$P/000006     377
*
```

If $P is not specified, its value is seven.

Breakpoints may be set in routines at different priority levels. For example, a program running at a low priority level may use a device service routine operating at a higher priority level. If a breakpoint occurs from a low priority routine, if ODT operates at a low priority, and if an interrupt does occur from a high priority routine, then the breakpoints in the high priority routine will not be executed since they have been removed.

5.3 ODT-11X

ODT-11X has all the commands and features of ODT-11 as explained in section 5.2, plus the following.

5.3.1 Opening, Changing and Closing Locations

In addition to operating on words, ODT-11X operates on bytes.

One way to open a byte is to type the address of the byte followed by a backslash:

```
*1001/025            (\ is printed by typing SHIFT and L)
```

A backslash typed alone reopens the last open byte. If a word was previously open, the backslash reopens its even byte.

```
*1002/000004\004
```

The LINE FEED and up-arrow (or circumflex) keys operate on bytes if a byte is open when the command is given. For example:

```
*1001\025 ↓
001002\004 ↑
001001\025
*
```

5.3.1.1 Open the Addressed Location(@) - The symbol @ optionally modifies, closes an open word, and uses its contents as the address of the location to open next.

```
*1006/001024   @               (open location 1024 next)
001024/000500
*1006/001024   2100 @          (modify to 2100 and open
002100/177774                   location 2100)
```

5.3.1.2 Relative Branch Offset(>) - The right angle bracket, >, optionally modifies, closes an open word, and uses its even byte as a relative branch offset to the next word opened.

```
*1032/000407   301  >          (modify to 301 and interpret
000636/000010                   as a relative branch)
```

Note that 301 is a negative offset (-77). The offset is doubled before it is added to the PC; therefore, 1034 + -176 = 636.

5.3.1.3 Return to Previous Sequence(<) - The left angle bracket, <, optionally modifies, closes an open location, and opens the next location of the previous sequence interrupted by a ←, @, or > command. Note that ← , @, > cause a sequence change to the word opened. If a sequence change has not occurred, < simply opens the next location as a LINE FEED does. The command operates on both words and bytes.

```
*1032/000407  301  >          (> causes a sequence change)
000636/000010 <               (<causes a return to original
                               sequence)
001034/001040 @               (@ causes a sequence change)
001040/000405\005 <           (< now operates on byte)
001035\ 002 <                 (< acts like ↓ )
001036\ 004
```

5.3.2 Calculating Offsets(n;O)

The command n;O causes ODT to print the 16-bit and 8-bit offsets from the currently open location to address n. The following examples, repeated from the ODT-11 section describing this command (see section 5.2.5), show a difference only in printout format:

```
*346/000034  414;O  000044   022   22
*/000022

*1034/103421  1034;O   177776  377\021   377
*/103777
```

Note that the modified low-order byte 377 must be combined with the unmodified high-order byte.

5.3.3 Breakpoints

With ODT-11X you can set up to eight breakpoints concurrently, numbered 0 through 7. The n;B command used in ODT-11 to set the breakpoint at address n sets the next available breakpoint in ODT-11X. Specific breakpoints may be set or changed by the n;mB command where m is the number of the breakpoint. For example:

```
*1020;B                     (sets breakpoint 0)
*1030;B                     (sets breakpoint 1)
*1040;B                     (sets breakpoint 2)
*1032;1B                    (resets breakpoint 1)
*
```

The ;B command used in ODT-11 to remove the only breakpoint removes all breakpoints in ODT-11X. To remove only one of the breakpoints, use the ;nB command, where n is the number of the breakpoint. For example:

```
*;2B                        (removes the second breakpoint)
*
```

The $B/ command opens the location containing the address of breakpoint 0. The next seven locations contain the addresses of the other breakpoints in order, and thus can be opened using the LINE FEED key. (The next location is for single-instruction mode, explained in the next section.) Example:

```
*$B/001020  ↓
nnnnnn/001032 ↓
nnnnnn/(address internal to ODT)
```

In this example, breakpoint 2 is not set. The contents are an address internal to ODT. After the table of breakpoints is the table of Proceed command repeat counts for each breakpoint and for the single-instruction mode (see Section 5.3.4).

```
.
.
.                ↓
nnnnnn/001036    ↓            (address of breakpoint 7)
nnnnnn/nnnnnn    ↓            (single-instruction address)
nnnnnn/000000    15 ↓         (count for breakpoint 0)
nnnnnn/000000                 (count for breakpoint 1)
```

It should be noted that a repeat count in a Proceed command refers only to the most recent breakpoint. Execution of other breakpoints encountered is determined by their own repeat counts.

5.3.4 Single-Instruction Mode

With this mode you can specify the number of instructions you wish executed before suspension of the program run. The Proceed command, instead of specifying a repeat count for a breakpoint encounter, specifies the number of succeeding instructions to be executed. Note that breakpoints are disabled when single-instruction mode is operative. Commands for single-instruction mode follow:

;nS   Enables single-instruction mode (n can have any value and serves only to distinguish this form from the form ;S); breakpoints are disabled.

n;P Proceeds with program run for next n instructions before reentering ODT (if n is missing, it is assumed to be 1). (Trap instructions and associated handlers can affect the Proceed repeat count. See section 5.5.2.)

;S Disables single-instruction mode

When the repeat count for single-instruction mode is exhausted and the program suspends execution, ODT prints:

```
B8;n
*
```

where n is the address of the next instruction to be executed. The $B breakpoint table contains this address following that of breakpoint 7. However, unlike the table entries for breakpoints 0-7, the B8 entry is not affected by direct modification.

Similarly, the repeat count for single-instruction mode follows the repeat count for breakpoint 7. This table entry, however, may be directly modified, and thus is an alternative way of setting the single-instruction mode repeat count. In such a case, ;P implies the argument set in the $B repeat count table rather than the argument 1.

5.4 ERROR DETECTION

ODT-11 and ODT-11X inform you of two types of errors: illegal or unrecognizable command and bad breakpoint entry.

Neither ODT-11 nor ODT-11X checks for the legality of an address when commanded to open a location for examination or modification.

Thus, the command

```
177774/
```

references nonexistent memory, and causes a trap through the vector at location 4. If this vector has not been properly initialized (by IOX, or the user program if IOX is not used), unpredictable results occur.

Similarly, a command such as

```
$20/
```

which references an address eight times the value represented by $2, may cause an illegal (nonexistent) memory reference.

Typing other than a legal command causes ODT to ignore the command, print

```
?
*
```

and wait for another command. Therefore, to cause ODT to ignore a command just typed, type an illegal character (such as 9 or RUBOUT) and the command will be treated as an error, i.e., ignored.

ODT suspends program execution whenever it encounters a breakpoint, i.e., a trap to its breakpoint routine. If the breakpoint routine is entered and no known breakpoint caused the entry, ODT prints:

```
BE001542
*
```

and waits for another command. In the example above, BE001542 denotes Bad Entry from location 001542. A bad entry may be caused by an illegal trace trap instruction, setting the T-bit in the status register, or by a jump to the middle of ODT.

5.5 PROGRAMMING CONSIDERATIONS

Information in this section is not necessary for the efficient use of ODT. However, its content does provide a better understanding of how ODT performs some of its functions.

5.5.1 Functional Organization

The internal organization of ODT is almost totally modularized into independent subroutines. The internal structure consists of three major functions: command decoding, command execution, and various utility routines.

The command decoder interprets the individual commands, checks for command errors, saves input parameters for use in command execution, and send control to the appropriate command execution routine.

The command execution routines take parameters saved by the command decoder and use the utility routines to execute the specified command. Command execution routines exit either to the object program or back to the command decoder.

The utility routines are common routines such as SAVE-RESTORE and I/O. They are used by both the command decoder and the command executers.

Communication and data flow are illustrated in Figure 5-1.

5.5.2 Breakpoints

The function of a breakpoint is to pass control to ODT whenever the user program tries to execute the instruction at the selected address. Upon encountering a breakpoint, the user can utilize all of the ODT commands to examine and modify his program.

When a breakpoint is executed, ODT removes the breakpoint instruction(s) from the user's code so that the locations may be examined and/or altered. ODT then types a message to the user, in the form Bn(Bm;n for ODT-11X), where n is the breakpoint address (and m is the breakpoint number). The breakpoints are automatically restored when execution is resumed.

A major restriction in the use of breakpoints is that the word

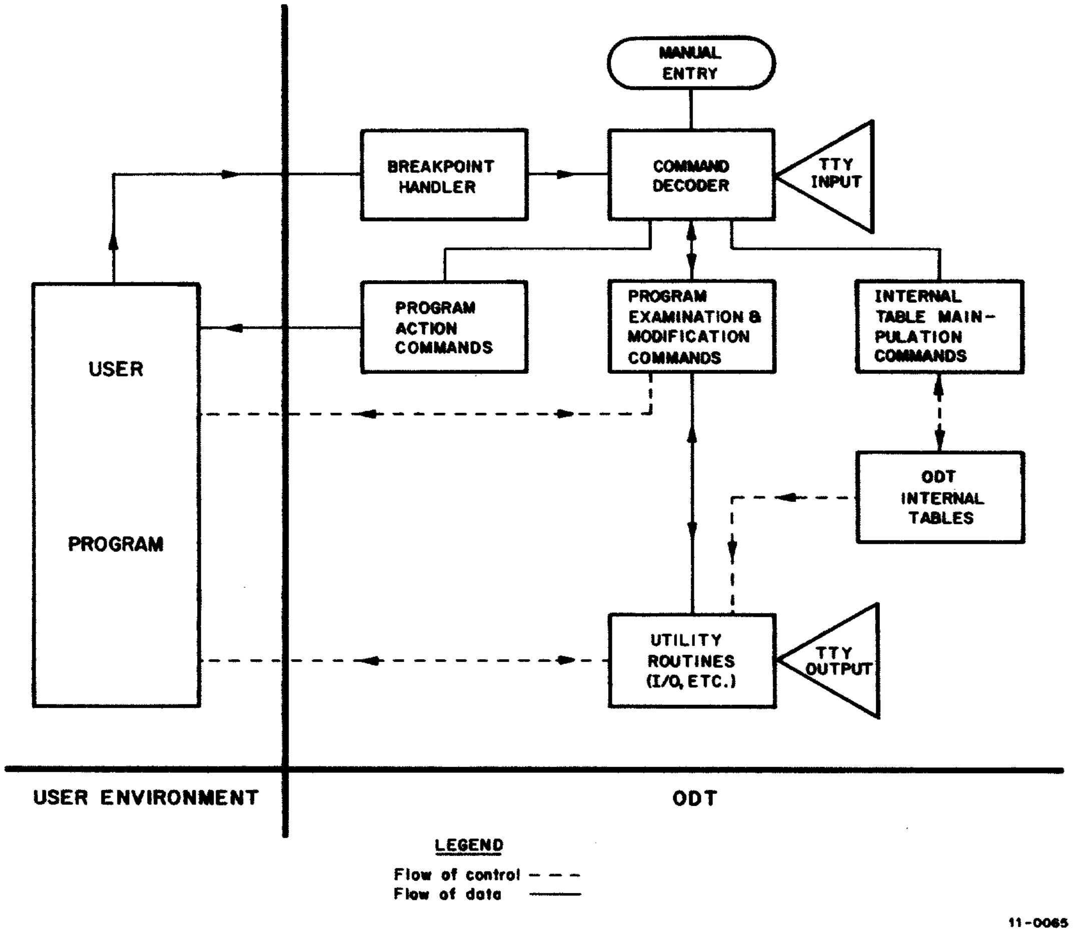


Figure 5-1 Communication and Data Flow

where a breakpoint has been set must not be referenced by the program in any way since ODT has altered the word. Also, no breakpoint should be set at the location of any instruction that clears the T-bit. For example:

```
     MOV #240,177776          ;SET PRIORITY TO LEVEL 5.
```

A breakpoint occurs when a trace trap instruction (placed in the user program by ODT) is executed. When a breakpoint occurs, ODT takes the following steps:

1. Set processor priority to seven (automatically set by trap instruction).

2. Save registers and set up stack.

3. If internal T-bit trap flag is set, go to step 13.

4. Remove breakpoint(s).

5. Reset processor priority to ODT's priority or user's priority.

6. Make sure a breakpoint or Single-instruction mode caused the interrupt.

7. If the breakpoint did not cause the interrupt, go to step 15.

8. Decrement repeat count.

9. Go to setp 18 if non-zero, otherwise reset count to one.

10. Save Teletype status.

11. Type message to user about the breakpoint or Single-instruction mode interrupt.

12. Go to command decoder.

13. Clear T-bit in stack and internal T-bit flag.

14. Jump to the "GO" processor.

15. Save Teletype status.

16. Type "BE" (Bad Entry) followed by the address.

17. Clear the T-bit, if set, in the user status and proceed to the command decoder.

18. Go to the "Proceed", bypassing the TTY restore routine.

Note that steps 1-5 inclusive take approximately 100 microseconds during which time interrupts are not permitted to occur (ODT is running at level 7).

When a proceed (;P) command is given, the following occurs:

1. The proceed is checked for legality.

2. The processor priority is set to seven.

3. The T-bit flags (internal and user status) are set.

4. The user registers, status, and Program Counter are restored.

5. Control is returned to the user.

6. When the T-bit trap occurs, steps 1, 2, 3, 13, and 14 of the breakpoint sequence are executed, breakpoints are restored, and program execution resumes normally.

When a breakpoint is placed on an IOT, EMT, TRAP, or any instruction causing a trap, the following occurs:

1. When the breakpoint occurs as described above, ODT is entered.

2. When ;P is typed, the T-bit is set and the IOT, EMT, TRAP, or other trapping instruction is executed.

3. The current PC and status (with the T-bit included) are pushed on the stack.

4. The new PC and status (no T-bit set) are obtained from the respective trap vector.

5. The whole trap service routine is executed without any breakpoints.

6. When an RTI is executed, the saved PC and PS (including the T-bit) are restored. The instruction following the trap-causing instruction is executed. If this instruction is not another trap-causing instruction, the T-bit trap occurs, causing the breakpoints to be reinserted in the user program, or the Single-instruction mode repeat count to be decremented. If the following instruction is a trap-causing instruction, this sequence is repeated, starting at step 3.

NOTE

Exit from the trap handler must be via the RTI instruction. Otherwise, the T-bit will be lost. ODT will not gain control again since the breakpoints have not been reinserted yet.

In ODT-11, the ;P command is illegal if a breakpoint has not occurred (ODT responds with ?). In ODT-11X, ;P is legal after any trace trap entry.

WARNING

Since ODT-11 ignores all semicolons, typing the ODT-11X form of breakpoint command number to ODT-11, specifying a breakpoint number n, causes the following error:

100;B (sets the breakpoint at location 100)
100;0B (sets the breakpoint at location 1000)
100;4B (sets the breakpoint at location 1004)

The internal breakpoint status words for ODT-11 have the following format:

1. The first word contains the breakpoint address. If this location points to a location within ODT, it is assumed no breakpoint is set for the cell (specifically, ODT has set a dummy breakpoint within itself).

2. The next word contains the breakpoint repeat count.

For ODT-11X (with eight breakpoints) the formats are:

1. The first eight words contain the breakpoint addresses for breakpoints 0-7. (The ninth word contains the address of the next instruction to be executed in Single-instruction mode.)

2. The next eight words contain the respective repeat counts. (The following word contains the repeat count for Single-instruction mode.)

The user may change these words at will, either by using the breakpoint commands or by direct manipulation with $B.

When program runaway occurs (that is, when the program is no longer under ODT control, perhaps executing an unexpected part of the program where a breakpoint has not been placed) ODT may be given control by pressing the HALT key to stop the machine, and restarting ODT (see Section 5.6.2). ODT prints *, indicating that it is ready to accept a command.

If the program being debugged uses the terminal for input or output, the program may interact with ODT to causes an error since ODT also uses the terminal. This interactive error does not occur when the program being debugged is run without ODT.

1. If the terminal output interrupt is enabled upon entry to the ODT break routine, and no output interrupt is pending when ODT is entered, ODT is entered, ODT generates an unexpected interrupt when returning control to the program.

2. If the interrupt of the terminal input (the keyboard) is enabled upon entry to the ODT break routine, and the program is expecting to receive an interrupt to input a character, both the expected interrupt and the character will be lost.

3. If the terminal input (keyboard) has just read a character into the reader data buffer when the ODT break routine is entered, the expected character in the input data buffer will be lost.

5.5.3 Search

The word search allows the user to search for bit patterns in specified sections of memory. Using the $M/ command, the user specifies a mask, a lower search limit ($M+2), and an upper search limit ($M+4). The search object is specified in the search command itself.

The word search compares selected bits (where ones appear in the mask) in the word and search object. If all selected bits are equal, ODT prints the unmasked word.

The search algorithm is:

1. Fetch a word at the current address.
2. XOR (exclusive OR) the word and search object.
3. AND the result of step 2 with the mask.
4. If the result of step 3 is zero, type the address of the unmasked word and its contents. Otherwise, proceed to step 5.
5. Add two to the current address. If the current address is greater than the upper limit, type * and return to the command decoder, otherwise go to step 1.

Note that if the mask is zero, ODT prints every word between the limits, since a match occurs every time (i.e., the result of step 3 is always zero).

In the effective address search, ODT interprets every word in the search range as an instruction which is interrogated for a possible direct relationship to the search object.

The algorithm for the effective address search is (where (x) denotes contents of x, and k denotes the search object):

1. Fetch a word at the current address X.
2. If (x)=k [direct reference], print contents and go to step 5.
3. If (x)+x+2=k [indexed by PC], print contents and go to step 5.
4. If (x) is a relative branch to k, print contents.
5. Add two to the current address. If the current address is greater than the upper limit, perform a carriage return/line feed and return to the command decoder; otherwise, go to step 1.

5.5.4 Teletype Interrupt

Upon entering the TTY SAVE routine, the following occurs:

1. Save the LSR status register (TKS).
2. Clear interrupt enable and maintenance bits in the TKS.
4. Clear interrupt enable and maintenance bits in the TPS.

To restore the TTY:

1. Wait for completion of any I/O from ODT.
2. Restore the TKS.
3. Restore the TPS.

NOTES

If the TTY printer interrupt is enabled upon entry to the ODT break routine, the following may occur:

1. If no output interrupt is pending when ODT is entered, an additional interrupt always occurs when ODT returns control to the user.

2. If an output interrupt is pending upon entry, the expected interrupt occurs when the user regains control.

If the TTY reader (keyboard) is busy or done, the expected character in the reader data buffer will be lost.

If the TTY reader (keyboard) interrupt is enabled upon entry to the ODT break routine, and a character is pending, the interrupt (as well as the character) will be lost.

5.6 OPERATING PROCEDURES

This section describes procedures for linking ODT on LSI-11 machines, and for loading ODT on other PDP-11 machines. It describes starting, restarting, error recovery, and setting the priority level of ODT.

5.6.1 Linking Procedures (LSI-11 Systems Only)

For LSI-11 systems, ODT-11X is supplied on relocatable object tapes. Binary tapes are produced by linking the ODT-11X object tape with the object tapes of the program to be debugged (using LINK-11S). The ODT-11X tape should be the first tape processed by LINK-11S; in this manner, ODT-11X is started first when the binary tape is loaded.

5.6.2 Loading Procedures (non-LSI-11 Systems Only)

For all systems other than LSI-11, ODT is supplied on source and binary tapes. Appendix N explains assembly instructions for source tapes. Binary tapes are loaded with the Absolute Loader. Since ODT is started as soon as it is loaded, the program to be debugged should be loaded prior to ODT.

When supplied on binary tape, ODT-11 loads beginning at location 13026, and occupies about 533 (decimal) words of memory. ODT-11X loads beginning at location 12054, and requires about 800 (decimal) words of memory.

5.6.3 Starting and Restarting

The Absolute Loader starts ODT automatically after loading it into core. ODT indicates its readiness to accept input by printing an *.

The starting address for ODT-11 on binary tape is 13026; the starting address for ODT-11X on binary tape is 12054. If ODT is reassembled using PAL-11A, the starting address in indicated in the symbol table as the value of the symbol O.ODT. If ODT is linked using LINK-11S, the starting address is indicated in the link map as the value of the global symbol O.ODT.

When ODT is started at its start address, the SP register is set to an ODT internal stack, registers R0-R5 are left untouched, and the trace trap vector is initialized. If ODT is started after breakpoints have been set in a program, ODT ignores the breakpoints and leaves the program modified, i.e., the breakpoint instructions are left in the program.

There are two ways to restart ODT:

1. Restart at start address+2

2. Reenter at start address+4

To restart, key in the start address+2, press LOAD ADDRess and then START. A restart saves the general registers, removes all the breakpoint instructions from the user program and then ignores all breakpoints, i.e., simulates the ;B command.

To reenter, key in the load address+4, press LOAD ADDRess and then START. A reenter saves the general registers, removes the breakpoint instructions from the user program, and types the BE (Bad Entry) error message. ODT remembers which breakpoints were set and resets them on the next ;G command (;P is illegal after a Bad Entry).

CHAPTER 6

LOADING AND DUMPING MEMORY

This chapter describes procedures for loading programs into memory (using the Bootstrap Loader and Absolute Loader) and for dumping the contents of memory (using the DUMPAB and/or DUMPTT programs).

The Bootstrap Loader, which loads short paper tape programs (162 or fewer octal words), appears on one of three forms, depending upon the system configuration:

1. Hardware - on some CPUs, the Bootstrap Loader is present as a ROM chip.

2. Software - on some CPUs, the Bootstrap Loader must be toggled in via console switches.

3. Firmware - on LSI-11s, the Bootstrap Loader is a firmware loader, present as a programmable ROM chip.

Once familiar with the operation of the Bootstrap Loader, the user can load other programs (such as the Absolute Loader, DUMPAB, and DUMPTT).

The Absolute Loader (see section 6.2) is a system program that enables the user to load data punched on paper tape in absolute binary format into any available memory bank. It is used primarily to load the paper tape system software, binary programs assembled with PAL-11A, and binary tapes produced by LINK-11S from object tapes produced by PAL-11S.

The loader programs are loaded into the upper-most area of available core and are available for use with system and user programs. Programs should not use the locations used by the loaders without restoring their contents; otherwise, the loaders must be reloaded since they will have been altered by the object program.

Core memory dump programs (see section 6.3) print or punch the contents of specified areas of core. For example, when developing or debugging user programs it is often necessary to get a copy of the program or portions of core. There are two dump programs supplied in the paper tape software system: DUMPTT, which prints or punches the octal representation of specified portions of core, and DUMPAB, which punches specified portions of core in absolute binary format suitable for loading with the Absolute Loader.

6.1 PAPER TAPE BOOTSTRAPS

Procedures for operating the various PDP-11 paper tape bootstraps are described below:

6.1.1 BM792-YA Paper Tape Bootstrap ROM

1. Set the console ENABLE/HALT switch to HALT.
2. Place the bootstrap tape in the desired paper tape reader with the special bootstrap leader code over the reader sensors (under the reader station).
3. If the low-speed reader (ASR-33) is to be used, and a high-speed reader is, present on the system, turn the high speed reader OFF. If the high-speed reader is to be used, turn it ON.
4. Set the console ENABLE/HALT switch to ENABLE.
5. Set the console switch register to 773000.
6. Press the console START switch. The contents of the bootstrap tape will be loaded into the highest locations of memory.
7. The bootstrap transfers control to the program just loaded. Typically, this program halts.

6.1.2 BM873-YA Bootstrap Loader ROM

1. Set the console ENABLE/HALT switch to HALT.
2. Place the bootstrap tape in the desired paper tape reader with the special bootstrap leader code over the reader sensors (under the reader station).
3. If the low-speed reader (ASR-33) is to be used, and a high-speed reader is present on the system, turn the high-speed reader OFF. If the high-speed reader is to be used, turn it ON.
4. Set the console ENABLE/HALT switch to ENABLE.

5a. If the low-speed reader is to be used, set the console switch register to 773210.

5b. If the high-speed reader is to be used, set the console switch register to 773312.

6. Press the console START switch. The contents of the bootstrap tape will be loaded into the highest locations of memory.
7. The bootstrap transfers control to the program just loaded. Typically, this program halts.

6.1.3 LSI-11 Firmware Paper Tape Bootstrap

1. Press the front panel BOOT/INIT switch. This enables the micro-ODT; an @ prints at the terminal.

2. Place the bootstrap tape in the desired paper tape reader with the special bootstrap leader code over the reader sensors (under the reader station).

3. If the low-speed reader (ASR-33) is to be used, and a high-speed reader is present on the system, turn the high-speed reader OFF. If the high-speed reader is to be used, turn it ON.

4. Type the command/status register address of the input device followed by L to load the tape.

   For example, when loading from the console terminal reader, type:

   ```
   @ 177560L
   ```

   After reading the contents of the tape, the LSI-11 microprocessor starts the program, which typically halts. In this case, the micro-ODT automatically restarts and prints @ followed by the address of the instruction after the HALT instruction. For example, after loading the Absolute Loader on an 8K system, the micro-ODT prints:

   ```
   @375000
   @
   ```

   The starting address of the Absolute Loader in this case is 375000.

6.1.4 M9301-YB Bootstrap Loader

1a. If the system does not have a switch register, press the front panel BOOT/INIT switch.

1b. If the system does not have a BOOT/INIT switch, set the console switch register to 773000; press LOAD/ADDR; then press START.

2. Four numbers are printed at the terminal, followed by a $. These numbers are the contents of the general registers R0, R4, R6, and R5, respectively. For CPUs without switch registers (such as the 11/04), R5 contains the contents of the program counter (PC) at the time BOOT/INIT was pressed.

   For example:

   ```
   007740  012450  00546  004054
   $
   ```

3. Place the bootstrap tape in the desired paper tape reader with the special bootstrap leader code over the reader sensors (under the reader station).

4. Type the device code (PR for high-speed reader, TT for terminal reader), and type RETURN, as follows:

   ```
   $PR↵    or    $TT↵
   ```

   After reading the contents of the tape, the Bootstrap Loader transfers control to the program just loaded. Typically, this program halts.

6.1.5 M9301-YA Bootstrap Loader

If a console terminal is available, boot instructions for the M9301-YA Bootstrap Loader are the same as for the M9301-YB Bootstrap Loader (Section 6.1.4).

If no console terminal is available, the auto-boot feature of the M9301-YA must be used. See the M9301 Maintenance Manual for instructions on placing the appropriate paper tape bootstrap in the M9301 module micro-switch. Then follow the procedure below:

1. Place the bootstrap tape in the desired paper tape reader with the special bootstrap leader code over the reader sensors (under the reader station).
2. Set the console HALT/CONT switch to CONT.
3. Press the console BOOT/INIT switch. After reading the contents of the tape, the Bootstrap Loader transfers control to the program just loaded. Typically, this program halts.

6.1.6 Other Bootstrap Loaders

This section is for users without any of the bootstrap aids listed above.

The Bootstrap Loader should be loaded (toggled) into the highest core memory bank. The locations and corresponding instructions of the Bootstrap Loader are listed and explained below.

Location	Instruction
xx7744	016701
xx7746	000026
xx7750	012702
xx7752	000352
xx7754	005211
xx7756	105711
xx7760	100376
xx7762	116162
xx7764	000002
xx7766	xx7400
xx7770	005267
xx7772	177756
xx7774	000765
xx7776	yyyyyy

Figure 6-1 Bootstrap Loader Instructions

In Figure 6-1, xx represents the highest available memory bank. For example, the first location of the Loader would be one of the following, depending on memory size, and xx in all subsequent locations would be the same as the first.

Location	Memory Bank	Memory Size
017744	0	4K
037744	1	8K
057744	2	12K
077744	3	16K
117744	4	20K
137744	5	24K
157744	6	28K

Note also in Figure 6-1 that the contents of location xx7766 should reflect the appropriate memory bank in the same manner as the location.

The contents of location xx7776 (yyyyyy in the Instruction column of Figure 6-1) should contain the device status register address of the paper tape reader to be used when loading the bootstrap formatted tapes. Either paper tape reader may be used, specified as follows:

Teletype Paper Tape Reader	--	177560
High-Speed Paper Tape Reader	--	177550

6.1.6.1 Loading the Loader Into Core - Toggle in the Bootstrap Loader as explained below.

1. Set xx7744 in the Switch Register (SR) and press LOAD ADDRess (xx7744 is displayed in the ADDRESS REGISTER).

2. Set the first instruction, 016701, in the SR and lift DEPosit (016701 is displayed in the DATA register).

NOTE

> When DEPositing data into consecutive words, the DEPosit automatically increments the ADDRESS REGISTER to the next word.

3. Set the next instruction, 000026, in the SR and lift DEPosit (000026 is displayed in the DATA register).

4. Set the next instruction in the SR, press DEPosit, and continue depositing subsequent instructions (ensure that location xx7766 reflects the proper memory bank) until after 000765 has been deposited in location xx7774.

5. Deposit the desired device status register address in location xx7776, the last location of the Bootstrap Loader.

It is good programming practice to verify that all instructions are stored correctly. This is done by proceeding at step 6 below.

6. Set xx7744 in the SR and press LOAD ADDRess.

7. Press EXAMine (the octal instruction in location xx7744 is displayed in the DATA register so that it can be compared to the correct instruction, 016701. If the instruction is correct, proceed to step 8; otherwise go to step 10.

8. Press EXAMine (the instruction of the location displayed in the ADDRESS REGISTER is displayed in the DATA register; compare the DATA register contents to the instruction for the displayed location.

9. Repeat step 8 until all instructions have been verified or go to step 10 whenever the correct instruction is not displayed.

When an incorrect instruction is displayed, it can be corrected by performing steps 10 and 11.

10. With the desired location displayed in the ADDRESS REGISTER, set the correct instruction in the SR and lift DEPosit (the contents of the SR are deposited in the displayed location).

11. Press EXAMine to ensure that the instruction was correctly stored (it is displayed in the DATA register).

12. Proceed at step 9 until all instructions have been verified.

The Bootstrap Loader is now loaded into core. The procedures above are illustrated in the flowchart of Figure 6-2.

6.1.6.2 Loading Bootstrap Tapes - Any paper tape punched in bootstrap format is referred to as a bootstrap tape (see Section 6.1.3) and is loaded into core using the Bootstrap Loader. Bootstrap tapes begin with about two feet of special bootstrap leader code (ASCII code 351, not blank leader tape as required by the Absolute Loader).

With the Bootstrap Loader in core, the bootstrap tape is loaded into core starting anywhere between location xx7400 and location xx7743, i.e., 162 (octal) words. The paper tape input device used is that which is specified in location xx7776 (see section 6.1.6.1).

Bootstrap tapes are loaded into core as explained below.

1. Set the ENABLE/HALT switch to HALT.

2. Place the bootstrap tape in the specified reader with the special bootstrap leader code over the reader sensors (under the reader station).

3. Set the console switch register to xx7744 (the starting address of the Bootstrap Loader) and press LOAD ADDRess.

4. Set the ENABLE/HALT switch to ENABLE.

5. Press START. The bootstrap tape passes through the reader as data is being loaded into core.

6. The bootstrap tape stops after the last frame of data (see Figure 6-5) has been read into core. The program on the bootstrap is now in core.

The procedures above are illustrated in the flowchart of Figure 6-3.

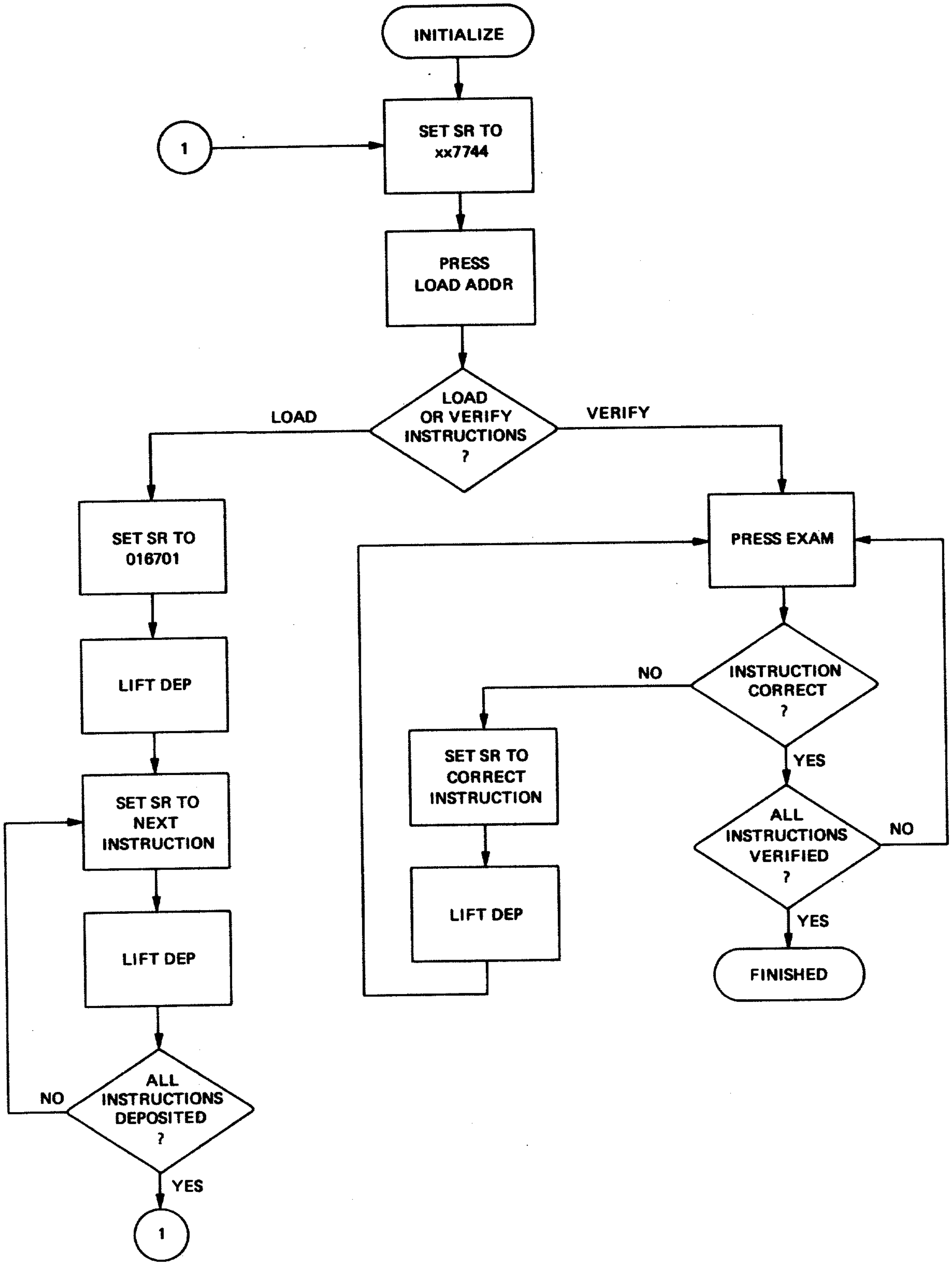


Figure 6-2 Loading and Verifying the Bootstrap Loader

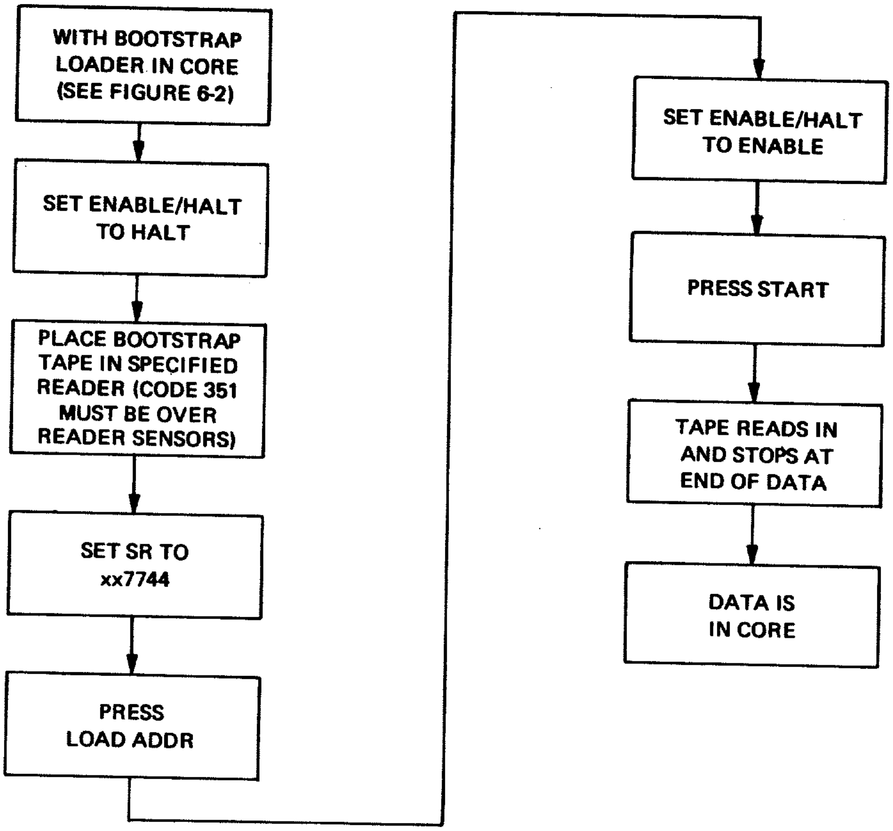

WITH BOOTSTRAP LOADER IN CORE (SEE FIGURE 6-2)
SET ENABLE/HALT TO HALT
PLACE BOOTSTRAP TAPE IN SPECIFIED READER (CODE 351 MUST BE OVER READER SENSORS)
SET SR TO xx7744
PRESS LOAD ADDR
SET ENABLE/HALT TO ENABLE
PRESS START
TAPE READS IN AND STOPS AT END OF DATA
DATA IS IN CORE

Figure 6-3 Loading Bootstrap Tapes Into Core

Should the bootstrap tape not read in immediately after depressing the START switch, one of the following conditions may exist:

1. Bootstrap Loader not correctly loaded.

2. Wrong input device used.

3. Code 351 not directly over the reader sensors.

4. Bootstrap tape not properly positioned in reader.

6.1.6.3 Bootstrap Loader Operation - The Bootstrap Loader source program is shown below. The starting address in the example denotes that the Loader is to be loaded into memory bank zero (a 4K system).

```
      000000           .ASECT
      000001 R1        =       %1        ;POINTER TO DEVICE ADDRESS
      000002 R2        =       %2        ;LOAD ADDRESS DISPLACEMENT
      000007 PC        =       %7        ;PROGRAM COUNTER
      017400 LOAD      =       17400     ;DATA CANNOT BE LOADED BELOW
                                         ;THIS ADDRESS

      017744 .         =       LOAD+344  ;STARTING ADDRESS

17744 016701 START:    MOV DEVICE,R1     ;COPY DEVICE ADDRESS
      000026
17750 012702 LOOP:     MOV (PC)+,R2      ;COPY ADDRESS DISPLACEMENT
17752 000352 DSPMNT:     +.-LOAD         ;INITIALLY OFFSET TO THIS LOC
                                         ;NOTE THAT THIS LOC IS PART OF
                                         ;PREVIOUS INSTRUCTION
17754 005211           INC @R1           ;START THE PAPER TAPE READER
17756 105711 WAIT:     TSTB @R1          ;FRAME READY?
17760 100376           BPL WAIT          ;BR IF NOT
17762 116162           MOVB 2(R1),LOAD(R2) ;STORE FRAME READ IN MEMORY
      000002
      017400
17770 005267           INC DSPMNT        ;INCREMENT DISPLACEMENT TO NEXT
      177756
                                         ;LOCATION
17774 000765 BRNCH:    BR LOOP           ;READ NEXT BYTE
17776 177560 DEVICE:   177560            ;ADDRESS OF INPUT DEVICE, MAY BE
                                         ;177550 IF HIGH SPEED READER


      000001'          .END
```

Figure 6-4 The Bootstrap Loader Program

The program above is a brief example of the PAL-11A Assembly Language which is explained in Chapter 2.

Bootstrap tapes are coded in the following format.

```
351
 .    Special bootstrap leader code (at least two feet
 .    in length)
351
xxx   Load offset (see text below)
AAA
BBB
CCC   Program to be loaded (up to 162  words or 344
 .    frames)
 .
 .
ZZZ
301
035
026
000
302   Boot overlay code, as shown.
025
373
yyy   Jump offset (see text below)
```

Figure 6-5 Bootstrap Tape Format

The Bootstrap Loader starts by loading the device status register address into R1 and 352_8 into R2. The next instruction indicates a read operation in the device and the next two instructions form a loop to wait for the read operation to be completed. When data is encountered it is transferred to a location determined by the sum of the index word (xx7400) and the contents of R2.

Because R2 is initially 352_8, the first word is moved to location xx7752, and it becomes the immediate data to set R2 in the next execution of the loop. This immediate data is then incremented by one and the program branches to the beginning of the loop.

The leader code, plus the increment, is equal in value to the data placed in R2 during the initialization; therefore, leader code has no effect on the loader program. Each time leader code is read the processor executes the same loop and the program remains unmodified. The first code other than leader code, however, replaces the data to be loaded into R2 with some other value which acts as a pointer to the program starting location (loading address). Subsequent bytes are read not into the location of the immediate data but into consecutive core locations. The program will thus be read in byte by byte. The INC instruction which operates on the data for R2 puts data bytes in sequential locations, and requires that the value of the leader code and the offset be one less than the value desired in R2.

The boot overlay code overlays the first two instructions of the Loader, because the last data byte is placed in the core location immediately preceding the Loader. The first instruction is unchanged by the overlay, but the second instruction is changed to place the next byte read, jump offset, into the lower byte of the branch instruction. By changing the offset in this branch instruction, the Loader can branch to the start of the loaded program or to any point within the program.

The Bootstrap Loader is self-modifying, and the program loaded by the Loader restores the Loader to its original condition by restoring the contents of locations xx7752 and xx7774 to 000352 and 000765 respectively.

6.2 THE ABSOLUTE LOADER

The Absolute Loader is a system program that enables the user to load data punched on paper tape in absolute binary format into any available memory bank. It is used primarily to load the paper tape system software, binary programs assembled with PAL-11A, and binary tapes produced by LINK-11S from object tapes produced by PAL-11S. The major features of the Absolute Loader include:

1. Testing of the checksum on the input tape to assure complete, accurate loads.

2. Starting the loaded program upon completion of loading without additional user action, as specified by the .END in the program just loaded.

3. Specifying the load bias of position independent programs at load-time rather than at assembly time, by using the desired Loader switch register option.

6.2.1 Loading the Loader Into Core

The Absolute Loader is supplied on punched paper tape in bootstrap format. Therefore, a Bootstrap Loader is used to load the Absolute Loader into core. It occupies locations xx7474 through xx7743, and its starting address is xx7500. The Absolute Loader program is 72 words long, and is loaded adjacent to the Bootstrap Loader as explained in section 6.1.6.2.

6.2.2 Using the Absolute Loader

Paper tapes punched in absolute binary format are also called absolute tapes, binary tapes, or .LDA tapes. These are the tapes loaded by the Absolute Loader.

In the following discussion, reference is made to a "switch register." For systems without switch registers (such as the LSI-11 and PDP-11/04), this term refers to a software switch register, which is a memory location internal to the Absolute Loader for systems without hardware switch registers. The location within the Absolute Loader is xxx516, where xxx reflects memory size as follows:

Memory	xxx
4K	017
8K	037
12K	057
16K	077
20K	117
24K	137
28K	157

When text indicates that a value be placed in a switch register, users without hardware switch registers must use either the M9301 console emulator or the LSI-11 micro-ODT, as appropriate, to store the switch register value in location xxx516. Once this value has been stored, the user starts the Absolute Loader at location xxx500. Once the Absolute Loader is loaded, it initializes the value of location xxx516 to 0. This value changes only when modified by the user.

A normal load occurs when data is loaded into memory according to the load addresses on the binary tape. The user must set bit 0 of the switch register to 0 immediately before starting the load.

There are two types of relocated loads:

1. Loading to continue from where the loader left off after the previous load -

   This is used, for example, when the object program being loaded is contained on more than one tape. It is specified by setting the switch register to 000001 immediately before starting the load.

2. Loading into a specific area of core -

   This is normally used when loading position independent programs. A position independent program is one which may be loaded and run anywhere in available core. The program is written using the position independent instruction format (see Chapter 9). This type of load is specified by setting the switch register to the load bias and adding 1 to it (i.e., setting bit 0 to 1). The effect of this is to add the value in the switch register to the start address on the tape.

Optional switch register settings for the three types of loads are listed below.

Type of Load	Switch Register Bits 1-14	 Bit 0
Normal	(ignored)	0
Relocated - continue loading where left off	0	1
Relocated - load in specified area of core	nnnnn (specified address)	1

The absolute tape may be loaded using either of the paper tape readers. The desired reader is specified in the last word of available core memory (xx7776), the input device status word, as explained in section 6.1.6. The input device status word may be changed at any time prior to loading the absolute tape.

With the Absolute Loader in core as explained in section 6.1.6.2, absolute tapes are loaded as explained below.

1. Set the ENABLE/HALT switch to HALT.

   To use an input device different from that used when loading the Absolute Loader, change the address of the device status word (in location xx7776) to reflect the desired device, i.e., 177560 for the Teletype reader or 177550 for the high-speed reader.

2. Set the switch register to xx7500 and press LOAD ADDR.

3. Set the switch register to reflect the desired type of load (Figure E-3 in Appendix E).

4. Place the absolute tape in the proper reader with blank leader tape directly over the reader sensors.

5. Set ENABLE/HALT to ENABLE.

6. Press START. The absolute tape begins passing through the reader station as data is being loaded into core.

If the absolute tape does not begin passing through the reader station, the Absolute Loader is not in core correctly. Reload the Loader and start over at step 1 above. If it halts in the middle of the tape, a checksum error occurred in the last block of data read in.

Normally, the absolute tape stops passing through the reader station when it encounters the transfer address as generated by the statement, .END, denoting the end of a program. If the system halts after loading, check that the low byte of the DATA register is zero. If so, the tape is correctly loaded. If not zero, a checksum error (explained later) has occurred in the block of data just loaded, indicating that some data was not correctly loaded. Reload the tape starting at step 1 above.

When loading a continuous relocated load, subsequent blocks of data are loaded by placing the next tape in the appropriate reader and pressing the CONTinue switch.

The Absolute Loader may be restarted at any time by starting at step 1 above.

6.2.3 Absolute Loader Operation

The Loader uses the eight general registers (R0-R7) and does not preserve or restore their previous contents. Therefore, caution should be taken to restore or load these registers when necessary after using the Loader.

A block of data punched on paper tape in absolute binary format has the following format.

FRAME	1	001	start frame
	2	000	null frame
	3	xxx	byte count (low 8 bits)
	4	xxx	byte count (high 8 bits)
	5	yyy	load address (low 8 bits)
	6	yyy	load address (high 8 bits)
		.	data is
		.	placed
		.	here
		zzz	last frame contains a block checksum

A program on paper tape may consist of one or more blocks of data. Each block with a byte count (frames 3 and 4) greater than six causes subsequent data to be loaded into core (starting at the address specified in frames 5 and 6 for a normal load). The byte count is a positive integer denoting the total number of bytes in the block, excluding the checksum. When the byte count of a block is six, the specified load address is checked to see whether the address is to an even or to an odd location. If even, the Loaded transfers control to the address specified. Thus the loaded program runs upon completion of loading. If odd, the loader halts.

The transfer address (TRA) may be explicitly specified in the source program by placing the desired address in the operand field following the .END statement. For example,

```
.END ALPHA
```

specifies the symbolic location ALPHA as the TRA, and

```
.END
```

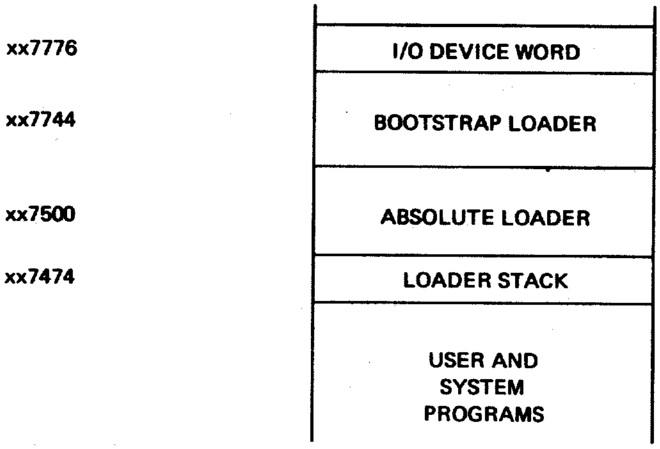
causes the Loader to halt. With

```
.END nnnnnn
```

the Loader also halts if the address (nnnnnn) is odd.

The checksum is displayed in the low byte of the DATA register of the computer console. Upon completion of a load, the low byte of the DATA register should be all zeros (unlit). Otherwise, a checksum error has occurred, indicating that the load was not correct. The checksum is the low-order byte of the negation of the sum of all the previous bytes in the block. When all bytes of a block including the checksum are added together, the low-order byte of the result should be zero.

If not, some data was lost during the load or erroneous data was picked up; the load was incorrect. When a checksum error is displayed, the entire program should be reloaded, as explained in the previous section. The loaders occupy core memory as illustrated below.



6.3 CORE MEMORY DUMPS

A core memory dump program is a system program which enables the user to dump (print or punch) the contents of any specified portion of core memory onto the Teletype printer and/or punch, line printer or high-speed punch. There are two dump programs available in the Paper Tape Software System:

1. DUMPTT[1], which dumps the octal representation of the contents of specified portions of core onto the teleprinter, low-speed punch, high-speed punch, or line printer.
2. DUMPAB, which dumps the absolute binary code of the contents of specified portions of core onto the low-speed punch or high-speed punch.

Both dump programs are supplied on punched paper tape in bootstrap and absolute binary formats. The bootstrap tapes are loaded over the Absolute Loader as explained in section 6.1.6.3, and are used when it would be undesirable to alter the contents of user storage (below the Absolute Loader). The absolute binary tapes are position independent and may be loaded and run anywhere in core as explained in section 6.2.2.

DUMPTT and DUMPAB are similar in function, and differ primarily in the type of output they produce.

6.3.1 Operating Procedures

Neither dump program punches leader or trailer tape, but DUMPAB always punches ten blank frames of tape at the start of each block of data dumped.

[1] DUMPTT is not available for systems without switch registers.

6.3.1.1 Using DUMPAB on Systems Without Switch Registers - Operating procedures for DUMPAB on systems without switch registers are as follows:

1. Select either the absolute binary or the bootstrap version of DUMPAB and place it in the reader specified by location xx7776 (see section 6.1).

2a. If using a bootstrap tape, load the tape using the procedure outlined in section 6.1. When the computer halts, go to step 3.

2b. If using an absolute binary tape, load the tape using the procedure outlined in section 6.2.2, relocating as follows:

   a. Select the address to which the program is to be relocated. The relocation offset is then equal to the loading address. For example, if the desired relocation address is 000400, the relocation offset is 000401.

   b. Deposit the relocation offset with bit 0 set in the Absolute Loader's software switch register. Using the example from the previous step, the user would deposit 000401 into location xxx516.

   Start the Absolute Loader.

3. When the program halts, find the address in the program counter. For LSI-11 machines, the value is printed at the console terminal by the micro-ODT. For UNIBUS PDP-11 machines, the user must press the BOOT/INIT switch to obtain register values at the console terminal (see section 6.1.4). The last of the four values displayed is the PC contents.

   Add 2 to the value of the PC. (For example, the PC contents for the bootstrap version of DUMPAB are xxx516; adding 2 to this value gives xxx520.) This new value is the address of the first of these succeeding parameters, described in subsequent steps.

4. Deposit the address of the first byte to be dumped into the first parameter (whose address was determined in the previous step).

5. Deposit the address of the last byte to be dumped into the second parameter (next sequential location).

6. The third parameter contains the value 177564 (a default specifying the ASR-33 punch). If this is the first time this step is executed and the high-speed reader is the desired output device, change the value of the third parameter to 177554.

7a. If using the LSI-11, type P to proceed.

7b. If using a UNIBUS PDP-11, restart the program (at xxx510 if bootstrap tape); press CONT when the program halts.

8. DUMPAB dumps the specified segment of memory and halts.

9. Repeat steps 4 through 8 until all desired memory segments have been dumped.

10. A transfer block for DUMPAB must be generated to terminate the dump. This value must be deposited in the first parameter (step 4) to terminate DUMPAB. If the tape is not to be self-starting, use 000001 as the transfer address. Under no conditions can 000000 be used as the transfer address.

11. Deposit 000000 in the second parameter (as in step 5).

12. Repeat step 7a or 7b, as appropriate, to punch the transfer block.

6.3.1.2 Using DUMPAB and DUMPTT on Systems with Switch Registers -

1. Select the dump program desired and place it in the reader specified by location xx7776 (see Section 6.1).

2. If a bootstrap tape is selected, load it using the Bootstrap Loader, section 6.1.6.2. When the computer halts go to step 4.

3. If an absolute binary tape is selected, load it using the Absolute Loader (section 6.2.2), relocating as desired.

   Place the proper start address in the switch register, press LOAD ADDRess and START. (The start addresses are shown in section 6.3.3).

4. When the computer halts, enter the address of the desired output device status register in the switch register and press CONTinue (low-speed punch and teleprinter = 177564; high-speed punch = 177554; line printer = 177514).

5. When the computer halts, enter in the switch register the address of the first byte to be dumped and press CONTinue. This address must be even when using DUMPTT.

6. When the computer halts again enter in the switch register the address of the last byte to be dumped and press CONTinue. When using the low-speed punch, set the punch to ON before pressing CONTinue.

7. Dumping proceeds on the selected output device.

8. When dumping is complete, the computer halts.

If further dumping is desired, proceed to step 5. It is not necessary to respecify the output device address except when changing to another output device. In such a case, proceed to the second paragraph of step 3 to restart.

If DUMPAB is being used, a transfer block must be generated as described below. If a tape read by the Absolute Loader does not have a transfer block, the loader will wait in an input loop. In such a case, the program may be manually initiated. However, this practice is not recommended, as there is no guarantee that load errors will not occur when the end of the tape is read.

The transfer block is generated by performing step 5 with the transfer address in the Switch Register, and step 6 with the transfer address minus 1 in the Switch Register. If the tape is not to be self-starting, an odd-numbered addrpss must be specified in step 5 (000001, for example).

The dump programs use all eight general registers and do not restore their original contents. Therefore, after a dump the general registers should be loaded as necessary prior to their use by subsequent programs.

6.3.2 Output Formats

The output from DUMPTT is in the following format:

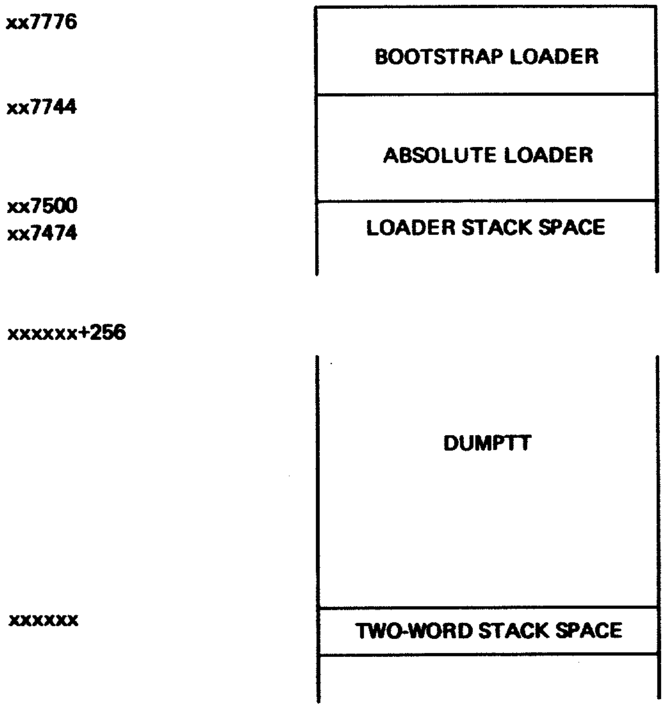
```
    xxxxxx>yyyyyy yyyyyy yyyyyy yyyyyy yyyyyy yyyyyy yyyyyy yyyyyy
```

where xxxxxx is the octal address of the first location printed or punched, and yyyyyy are words of data, the first of which starts at location xxxxxx. This is the format for every line of output. There will be no more than eight words of data per line, but there will be as many lines as are needed to complete the dump.

The output from DUMPAB is in absolute binary, as explained in section 6.2.3.

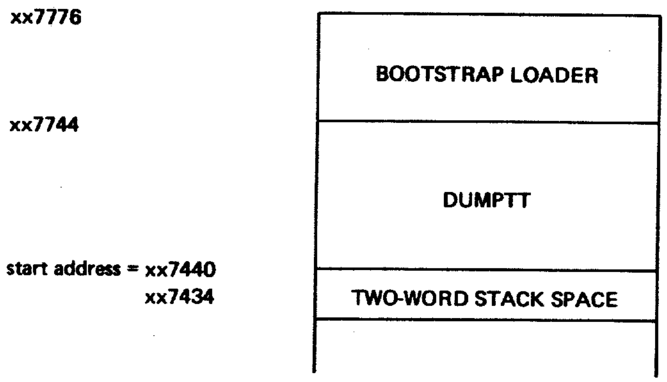
6.3.3 Storage Maps

The DUMPTT program is 87 words long. When used in absolute format the storage map is:

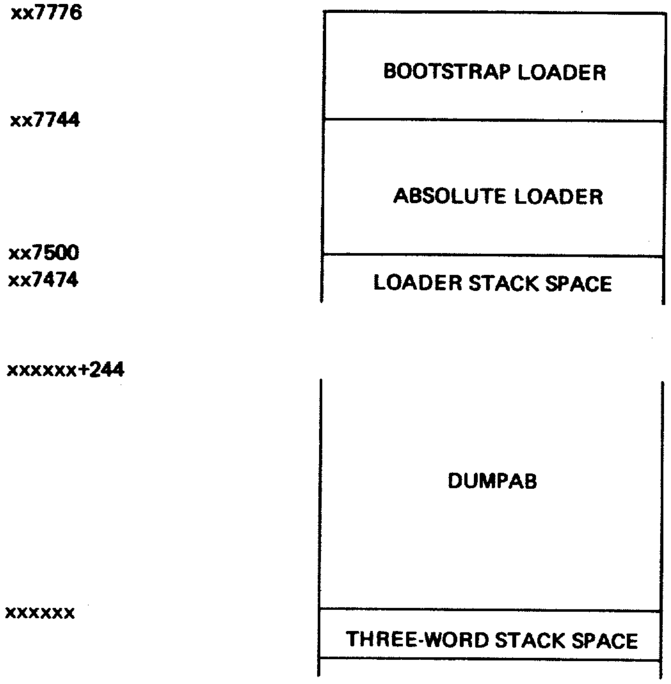


xxxxxx = desired load address = start address

When used in bootstrap format the storage map is:

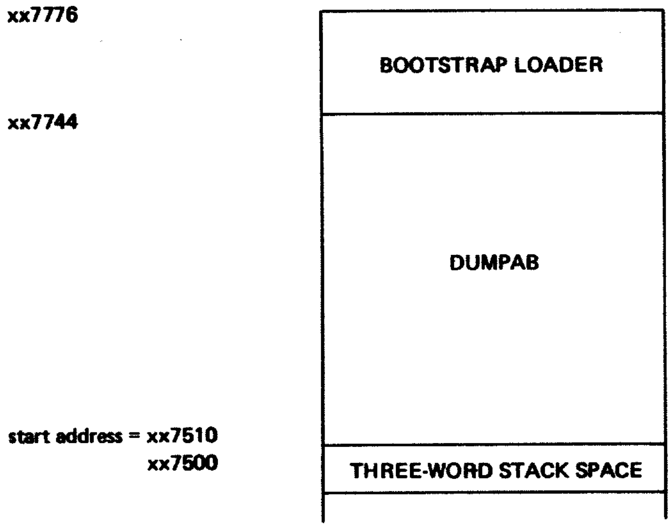


The DUMPAB program (for systems with a switch register) is 65(10) words long. When used in absolute format the storage map is:

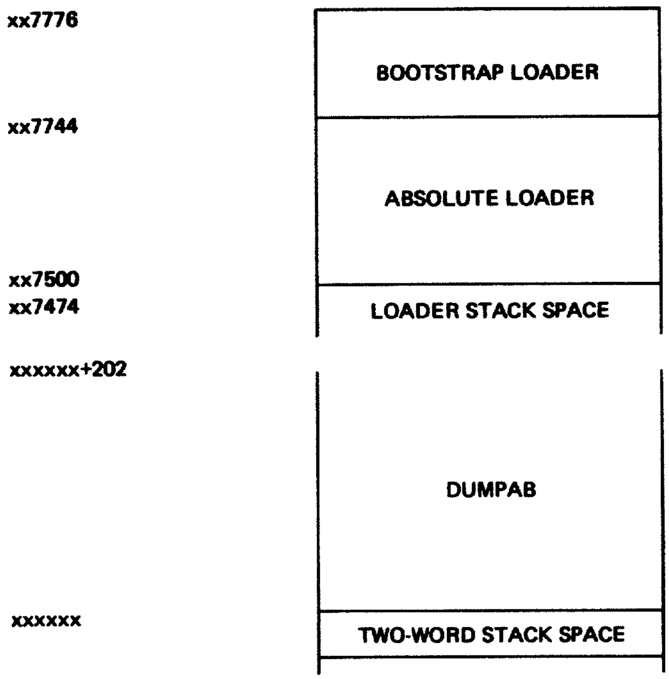


xxxxxx = desired load address = start address

When used in bootstrap format the storage map is:

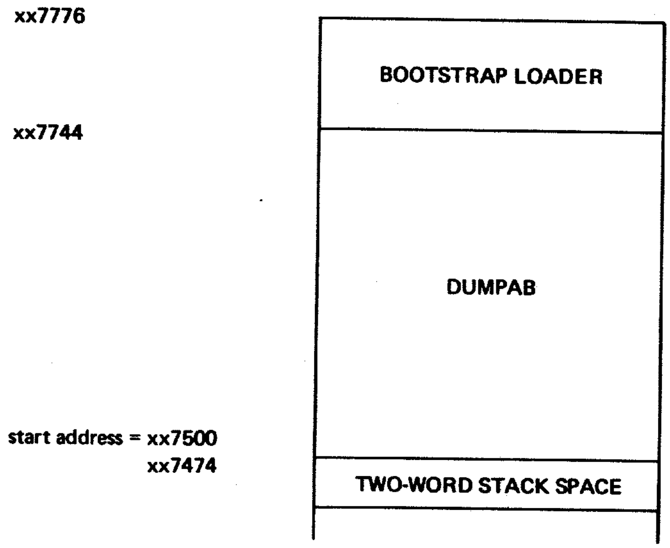


The DUMPAB program (for systems without a switch register) is 82(10) words long. When used in absolute format the storage map is:



xxxxxx = desired load address = start address

When used in bootstrap format the storage map is:



CHAPTER 7

INPUT/OUTPUT PROGRAMMING

7.1 INTRODUCTION

The PDP-11 Input/Output eXecutive (IOX), frees the user from dealing directly with I/O devices. It provides programming formats that allow programs written for the paper tape software system to be used later in a monitor environment with only minor coding changes.

IOX provides asynchronous I/O service for the following non-file-oriented devices:

1. Teletype keyboard, printer, and tape reader and punch
2. High-speed paper tape reader and punch

For line printer handling, in addition to all IOX facilities, IOXLPT is available.

Simple I/O requests can be made, specifying devices and data forms for interrupt-controlled data transfers, that can occur concurrently with the execution of a user program. Multiple I/O devices can run single- or double-buffered I/O processing simultaneously.

Real-time capability is provided by allowing user programs to be executed at device priority levels upon completion of a device action or data transfer.

Communication with IOX is accomplished by IOT (Input/Output Trap) instructions in the user's program. Each IOT is followed by two or three words consisting of one of the IOX commands and its operands. The IOX commands can be divided into two categories:

1. those concerned with establishing necessary conditions for performing input and output (mainly initializations), and
2. those concerned directly with the transfer of data.

When transfer of data is occurring, IOX is operating at the priority level of the device. The calling program runs at its priority level, either concurrent with the data transfer, or sequentially.

Programming format for commands is:

```
        IOT
        .WORD (an address)
        .BYTE (a command code), (a slot number)
```

Before using the data transfer commands, two preparatory tasks must be performed:

1. Since device specifications are made by referring to "slots" in IOX's Device Assignment Table (DAT) rather than devices themselves, the slots specified in the code must have devices assigned to them.

2. The buffer, whose address is specified in the code, must be set up with information about the data.

In those non-data-transfer commands where an address or slot number does not apply, a 0 must be used. Addresses or codes indicated can, of course, be specified symbolically.

The following program segment illustrates a simple input-process-output sequence. It includes:

1. The setting up of a single buffer

2. All necessary initializations

3. A formatted ASCII read into the buffer

4. A wait for completion of the read

5. Processing of data just read

6. A write command from the buffer.

```
          RESET=2               ;ASSIGN IOX COMMAND CODES
          READ=11
          WAITR=4
          WRITE=12

          IOT                   ;IOX RESET TO DO NECESSARY
          .WORD 0               ;INITIALIZATIONS INCLUDING
          .BYTE RESET,0         ;INITING SLOT 0 FOR KBD, AND 1 FOR TTY

          IOT                   ;TRAP TO IOX
          .WORD BUFFER          ;SPECIFY BUFFER
          .BYTE READ,0          ;READ FROM KBD (SLOT 0) TILL
                                ;LINE FEED OR FORM FEED

WAIT:     IOT                   ;TRAP TO IOX
          .WORD WAIT            ;BUSY RETURN ADDRESS WHILE WAITING
                                ;FOR KBD TO FINISH
          .BYTE WAITR,0         ;WAIT FOR KBD (SLOT 0) TO FINISH
          (process BUFFER)

          IOT                   ;TRAP TO IOX
          .WORD BUFFER          ;SPECIFY BUFFER
          .BYTE WRITE,1         ;WRITE TO TELEPRINTER (SLOT 1)

BUFFER:   100                   ;BUFFER SIZE IN BYTES
          0                     ;CODE FOR FORMATTED ASCII MODE
          0                     ;IOX WILL SET HERE THE NUMBER OF BYTES READ
          .=.+100               ;STORAGE RESERVED FOR 100 BYTES
```

In more complex programming it is likely that more than one buffer will be set up for the transfer of data, so that data processing can occur concurrently rather than sequentially, as here. Note too, that there are five IOX commands not used in this example that will help meet the requirements of I/O problems not as straightforward as this.

7.1.1 Using IOX With The LSI-11 Processor

IOX (IOXLPT) is supplied on source and relocatable object tapes. It is thus unnecessary to assemble IOX unless the program is to be modified. User object tapes can be linked with the IOX object tape (using LINK-11S) to produce an absolute binary tape. Appendix J describes assembly procedures for source tapes.

IOX requires approximately 633 (decimal) words of core; IOXLPT requires approximately 724 (decimal) words.

7.1.2 Using IOX with Unibus PDP-11 Processors

IOX (IOXLPT) is supplied on source and binary tapes. Appendix J describes assembly procedures for source tapes. Binary tapes are loaded prior to user programs by the Absolute Loader. After IOX is loaded, the Absolute Loader halts.

IOXLPT is used instead of IOX if the program uses a line printer.

IOX is supplied on an absolute binary tape with a loading address of 15100; the load address for IOXLPT is 34600. If the user desires different load addresses, the programs must be reassembled as described in Appendix J.

IOX requires approximately 634 (decimal) words of core; IOXLPT requires approximately 725 (decimal) words.

7.1.3 IOX Interrupt and Trap Vectors

IOX (IOXLPT) loads the following interrupt and trap vectors:

Console terminal
high speed reader and punch
timeout and other errors
IOT
line printer (IOXLPT only)

7.2 THE DEVICE ASSIGNMENT TABLE

The Device Assignment Table (DAT) makes programs device-independent by allowing the user to refer to a slot to which a device has been assigned, rather than a specific device itself. Thus, changing the input or output device becomes a simple matter of reassigning a different device to the slot indicated in the program.

The DAT is created by means of the Reset and/or Init commands. The IOX codes for devices (listed in the description of the Init command below) are assigned to the slots.

7.2.1 Reset

```
IOT
.WORD 0
.BYTE 2,0
```

This command must be the first IOX command issued by a user program. It clears the DAT, initializes IOX, resets all devices to their state at power-up, enables keyboard interrupts, and initializes DAT slots 0 and 1 for the keyboard and teleprinter, respectively.

7.2.2 Initialization

```
IOT
.WORD (address of device code)
.BYTE 1, (slot number)
```

The device whose code (stored as a byte) is found at the specified address is associated with the specified slot (numbered in the range 0-7). The device interrupt is turned off when necessary. (The keyboard interrupt always remains enabled.) There is no restriction on the number of slots that can be initialized to the same device.

DEVICE		DEVICE CODE
Teletype Keyboard	(KBD)	1
Teletype printer	(TTY)	2
Low-Speed Reader	(LSR)	3
Low-Speed Punch	(LSP)	4
High-Speed Reader	(HSR)	5
High-Speed Punch	(HSP)	6
Line Printer (IOXLPT only)	(LPT)	10

Note that a device code is used only in the Initialization (INIT) command. All other commands that refer to a device do so by means of a slot. Example:

```
           INIT=1
           IOT                   ;TRAP TO IOX
           .WORD HSRCOD          ;INIT SLOT 3
           .BYTE INIT,3          ;FOR HSR
            .
            .
            .
HSRCOD:    .BYTE 5               ;HSR CODE
```

7.3 BUFFER ARRANGEMENT IN DATA TRANSFER COMMANDS

Use of data-transfer commands (Read, Write, Real-time Read, Real-time Write) requires the creation of at least one buffer. This buffer is used not only to store data for processing, but to hold information regarding the quantity, form, and status of the data. The non-data portion of the buffer is called the buffer header, and precedes the data portion. In data transfer commands, it is the address of the first word of the buffer header that is specified in the word following the IOT of the command.

NOTE

> IOX uses the buffer header while transferring data. The user's program must not change or reference it.

The buffer format is:

	Location	Contents
BUFFER HEADER	Buffer	Maximum number of data bytes (unsigned integer)
	Buffer+2	Mode of data (byte)
	Buffer+3	Status of data (byte)
	Buffer+4	Number of data bytes involved in transfer (unsigned integer)
	Buffer+6	Actual data begins here

BUFFER SIZE (IN BYTES)	
STATUS	MODE
BYTE COUNT	
DATA . . .	

7.3.1 Buffer Size

The first word of the buffer contains the size (in bytes) of the data portion of the buffer as specified by the user. IOX will not store more than this many data bytes on input. Buffer size has no meaning on output.

7.3.2 Mode Byte

The low-order byte of the second word holds information concerning the mode or transfer. A choice of four modes exists:

		Coded as	
1.	Formatted ASCII	0	(or 200 to suppress echo)
2.	Formatted Binary	1	
3.	Unformatted ASCII	2	(or 202 to suppress echo)
4.	Unformatted Binary	3	

The term echo applies only to the KBD. Data transfers from other devices never involve an echo.

MODE BYTE

Bits	7	6	5	4	3	2	1	0	Bits
1=	NO ECHO						UNFOR-MATTED	BINARY	=1
0=	ECHO						FOR-MATTED	ASCII	=0

7.3.3 Status·Byte

The high-order byte of the second word of the buffer header contains information set by IOX on the status of the data transfer:

Bits 0-4 contain the non-fatal error codes (coded octally)

Bit 5 1 = End-Of-File has occurred (attempt at reading data after an End-Of-Medium)

Bit 6 1 = End-of-Medium has occurred (see Section 7.3.3.3)

Bit 7 1 = Done (Data Transfer complete)

STATUS BYTE

7	6	5	4	3	2	1	0
1 = DONE	1 = EOM	1 = EOF	SEE CODES				
			NON-FATAL ERRORS				

7.3.3.1 Non-Fatal Error Codes (Octal) -

2 = checksum error

3 = truncation of a long line

4 = an improper mode

1. A checksum error can occur only on a Formatted Binary read (see Section 7.4.3).

2. Truncation of a long line can occur on either a Formatted Binary or Formatted ASCII read (Section 7.4.1). This error occurs when the binary block or ASCII line is bigger than the buffer size specified in the buffer header. In both cases, IOX continues reading characters into the last byte in the buffer until the end of the binary block or ASCII line is encountered.

3. An improper mode can occur only on a Formatted Binary read. Such occurrence means that the first non-null character encountered was not the proper starting character for a Formatted Binary block (see Section 7.4.3)

7.3.3.2 Done Bit - When the data transfer to or from the buffer is complete, the Done Bit is set by IOX.

7.3.3.3 End-Of-Medium Bit - The following conditions cause the EOM bit to be set in the buffer Status byte associated with a data transfer command. An EOM occurrence also sets the Done Bit.

HSR	HSP	LSR	LPT
No tape	No tape	Timeout detected	No paper
Off line	No power		No power
No power			Printer drum gate open
			Overtemperature condition

An End-Of-Medium condition on an output device is cleared by a manual operation such as putting a tape in the high-speed punch. IOX does not retain any record of an EOM on an output device. However, an EOM on an input device is recorded by IOX so that succeeding attempts to read from that device will cause an End-Of-File (see Section 7.3.3.4). To reenable input the device must be manually readied and a Seek command (Section 7.6) executed on the proper slot. The INIT and RESET commands will also clear the EOM condition for the device.

See Section 7.5.3 for information on detection of conditions causing LSR timeouts.

When an End-Of-Medium has occurred on a Read, there may be data in the buffer. If an EOM has occurred on a Write, there is no way of knowing how much of the buffer was written.

7.3.3.4 End-Of-File Bit - An EOF condition appears in the Status byte if an attempt to read is made after an EOM has occurred. EOF cannot occur on output. When an EOF has occurred, no data is available in the buffer.

7.3.4 Byte Count

The third word contains the Byte Count:

Input: In unformatted data modes, IOX reads as many data bytes as the user has specified. In formatted modes, IOX inserts here the number of data bytes available in the buffer. In all modes, if an EOM occurs, IOX will set the Byte Count equal to the number of bytes actually read. If an EOF occurs, Byte Count will be set to 0.

Output: Byte Count determines the number of bytes output, for all modes. An HSP end-of-tape or LPT out-of-paper condition will also terminate output, and EOM will be set in the Status byte. IOX does not modify the Byte Count on output.

7.4 MODES

7.4.1 Formatted ASCII

A Formatted ASCII read transfers 7-bit characters (bit 8 will be zero) until a line feed or form feed is read. IOX sets the Byte Count word in the buffer header to indicate the number of characters in the buffer. If the line is too long, characters are read and overlaid into the last byte of the buffer until an end-of-line (a line feed or form feed) or EOM is detected. Thus, if there is no error, the buffer will always contain a line feed or form feed.

A Formatted ASCII write transfers the number of 7-bit characters specified by the buffer Byte Count. Bit 8 will always be output as zero.

Device-Dependent Functions

Keyboard

Seven-bit characters read from the keyboard are entered in the buffer and are echoed on the teleprinter except as follows:

Null - Ignored. This character is not echoed or transferred to the buffer.

Tab (CTRL/TAB keys) - Echoes as spaces up to the next tab stop. "Stops" are located at every 8th carriage position.

RUBOUT - Deletes the previous character on the current line and echoes as a backslash (\). If there are no characters to delete, RUBOUT is ignored.

CTRL/U - Deletes the current line and echoes as ↑U.

Carriage Return (RETURN key) - Echoes as a carriage return followed by a line feed. Both characters enter the buffer.

CTRL/P - Echoes as ↑P and causes a jump to the restart address, if non-zero (see 7.6.2).

The echo may be suppressed by setting bit 7 of the buffer header Mode byte.

If the buffer overflows, only the characters which fit into the buffer are echoed. Of course, characters which are deleted by RUBOUT or CTRL/U do not read into the buffer even though they are echoed. If a carriage return causes an overflow, or is typed after an overflow has occurred, a carriage return and line feed will be echoed but only the line feed will enter the buffer.

In the following Formatted ASCII examples:

1. assume there is room for five characters
2. ↵ indicates:
   in left column, the RETURN key
   in center column, the execution of a carriage return
   in right column, the ASCII code for carriage return
3. ↓ indicates:
   in center column, the execution of a line feed
   in right column, the ASCII code for line feed
4. RUB
OUT indicates the RUBOUT key
5. CTRL
U indicates the CTRL and U keys.

Typed	Echoed	Entered Buffer
ABC↵	ABC↵↓	ABC↵↓
ABCD↵	ABCD↵↓	ABCD↓
ABCDEF↵	ABCD↵↓	ABCD↓
ABCDEF RUB ↵ OUT	ABCD\↵↓	ABC↵↓
CTRL RUB ↵ U OUT	↑U↵↓	↵↓
ABCDEF RUB RUB ↵ OUT OUT	ABCD\\↵↓	AB↵↓
ABCDEF RUB RUB RUB x↵ OUT OUT OUT	ABCD \\\X↵↓	AX↵↓

Low-Speed Reader and High-Speed Reader

All characters are transferred to the buffer except that nulls and rubouts are ignored.

Teleprinter

Characters are printed from the buffer as they appear except that nulls are ignored and tabs are output as spaces up to the next tab stop.

Low-Speed Punch and High-Speed Punch

Characters are punched from the buffer as they appear except that nulls are ignored and tabs are followed by a rubout.

Line Printer (IOXLPT only)

Characters are printed from the buffer as they appear except as follows:

Nulls - Ignored

Tab - Output as spaces up to the next tab stop.

Carriage Return - Ignored. It is assumed that a line feed or form feed follows. These characters cause the line printer "carriage" to advance.

All characters beyond the 80th are ignored except a line feed or form feed.

7.4.2 Unformatted ASCII

Unformatted ASCII transfers the number of 7-bit characters specified by the header Byte Count.

Device-Dependent Functions

Keyboard

Characters are read and echoed except as follows;

Tab - Echoes as spaces up to the next tab stop.

CTRL/P - Echoes as ^P and causes a jump to the restart address, if non-zero (see 7.6.2).

7.4.3 Formatted Binary

Formatted Binary is used to transfer checksummed binary data (8-bit characters) in blocks. A Formatted Binary block appears as follows:

Byte (Octal)		Meaning
001	-	Start of block
000	-	Always null
XXX XXX	- -	Block Byte Count (low-order followed by high-order). Count includes data and preceding four bytes.
DDD DDD . . . . . DDD DDD	-	Data bytes
CCC	-	Checksum. Negation of the sum of all preceding bytes in the block.

IOX creates the block on output, from the buffer and buffer header. The Byte Count word in the buffer header specifies the number of data bytes following, which are to be output. Note that the Byte Count output is four larger than the header Byte Count. As the block is output, IOX calculates the checksum which is output following the last data byte.

On Formatted Binary reads, IOX ignores null characters until the first non-null character is read. If this character is a 001, a Formatted Binary block is assumed to follow and is read from the device under control of the Byte Count value. If the first non-null character is not 001, the read is immediately terminated and error code 4 is set in the Status byte. As the block is read a checksum is calculated and compared to the checksum following the block. If the checksum is incorrect, error code 2 is set in the Status byte of the buffer header. If the binary block is too large (Byte Count less 4, larger than the Buffer Size specified in the header), the last byte of the buffer is overlaid until the last data byte has been read; error code 3 is set in the Status byte.

Device-Dependent Functions

None. Eight-bit data characters are transferred to and from the device and buffer exactly as they appear.

7.4.4 Unformatted Binary

This mode transfers 8-bit characters with no formatting or character conversions of any kind. For both input and output, the buffer header Byte Count determines the number of characters transferred.

Device-Dependent Functions

None

7.5 DATA TRANSFERS

7.5.1 Read

```
IOT
.WORD (address of first word of the buffer header)
.BYTE 11, (slot number)
```

This command causes IOX to read from the device associated with the specified slot according to the information found in the buffer header. IOX initiates the transfer of data, clears the Status byte, and returns control to the calling program. If the device on the selected slot is busy, or a conflicting device (see Section 7.5.3) is busy, IOX retains control until the data transfer can be initiated. Upon completion of the Read, the appropriate bits in the Status byte are set by IOX and the Byte Count word indicates the number of bytes in the data buffer. Note that use of the KBD while an LSR Read is in progress will intersperse KBD characters into the buffer unpredictably.

7.5.2 Write

```
IOT
.WORD (address of first word of the buffer header)
.BYTE 12, (slot number)
```

IOX writes on the device associated with the specified slot according to the information found in the buffer header. Transfer of data occurs in the amount specified by Byte Count (Buffer+4). IOX returns control to the calling program as soon as the transfer has been initiated. If the device on the selected slot is busy, or a conflicting device is busy, IOX retains control until the transfer can be initiated. Upon completion of the Write, IOX will set the Status byte to the latest conditions. If a Write causes an EOM condition, the user has no way of determining how much of his buffer has been written (the Byte Count remains the same.)

7.5.3 Device Conflicts In Data Transfer Commands

Because there is a physical association between the devices on the ASR Teletype, certain devices cannot be in use at the same time. When a data transfer command is given, IOX simultaneously checks for two conditions before executing the command:

1. Is the device requested already in use? and,

2. Is there some other device in use that would result in an operational conflict?

IOX resolves both conflict situations by waiting until the first device is no longer busy, before allowing the requested device to start functioning. (This is an automatic Waitr command. See next section.) For example, if the LSR is in use, and either a KBD request or a second request for the LSR itself is made, IOX will wait until the current LSR read has been completed before returning control to the calling program. In the particular case of the LSR, IOX also performs a timeout check while waiting for it to become available.

When a Read command has been issued for the LSR, IOX waits about 100 milliseconds for each character to be read. If no character is detected by this time (presumably because the LSR is turned off, or out of tape), a timeout is declared and IOX sets EOM in the appropriate buffer Status byte.

The following is a table listing the devices. Corresponding to each device on the left is a list of devices (or the echo operation) which would conflict with it in operation.

Device	All Possible Conflicting Devices or Operations
KBD	CHO, KBD, TTY, LSR, LSP
TTY	Echo, KBO, TTY, LSP
LSR	KBD, LSR
LSP	Echo, KBD, TTY, LSP
HSR	HSR
HSP	HSP
LPT (IOXLPT only)	LPT

7.5.4 Waitr (Wait, Return)

```
IOT
.WORD (busy return address)
.BYTE 4, (slot number)
```

Waitr, like device conflict resolution, causes IOX to test the status of the device associated with the specified slot. If the device (or any possible conflicting device) is not transferring data, control is passed to the instruction following the Waitr. Otherwise, IOX transfers program control to the busy return address. If it is desired to continuously test for completion of data transfer on the device, the busy return address of the immediately preceding IOT instruction can be specified, effecting a Wait loop.

If a slot is inited to any device other than the LSR, control is returned to the calling program about 150 microseconds after execution of a Waitr. For the LSR, however, the time is about 100 milliseconds.

Note that a not-busy return from Waitr normally means the device is available. However, in the case of a Write, this only means that the last character has been output to the device. The device is still in the process of printing or punching the character. Thus, care must be exercised when performing an IOX Reset, hardware RESET, or HALT after a Write-Waitr sequence, since these may prevent the last character from being physically output.

7.5.5 Waitr vs. Testing the Buffer Done Bit

Since IOX permits you to have device-independent code, it may not be known, from run to run, what devices will be assigned to the slots in your program. Waitr tests the status, not only of the device it specifies, but also of all possible conflicting devices.

This means that when Waitr indicates that the device is not busy, the data transfer on the device of interest may have been done for some time. Depending on the program and what devices are assigned to the slots for a given run, the Waitr could have been waiting an additional amount of time for a conflicting device to become free.

Where this possibility exists and buffer availability is what is of interest, testing the Done bit of the Status byte (set when buffer transfer is complete) would be preferable to Waitr; whereas Waitr would be preferable if device availability is what is of interest.

This distinction is made in order to write device-independent code. In the example below:

1. If the devices at slots 2 and 3 could be guaranteed always to be conflicting, neither Waitr nor testing the Done bit would be necessary, because IOX would automatically wait for the busy device to finish before allowing the other device to begin.

2. If these devices could be guaranteed never to be conflicting, it wouldn't matter which of these methods was used, because Waitr couldn't be waiting extra time for a conflicting device (of no interest) to become free.

```
Example:                PROGRAM A                PROGRAM B

                     IOT                      IOT
                     .WORD BUF2               .WORD BUF2
                     .BYTE READ, SLOT2        .BYTE READ, SLOT2

                     IOT                      IOT
                     .WORD BUF1               .WORD BUF1
                     .BYTE READ, SLOT2        .BYTE READ, SLOT2

                     IOT                      IOT
                     .WORD BUF2               .WORD BUF2
                     .BYTE WRITE, SLOT3       .BYTE WRITE, SLOT3

      DUNTST:        TSTB BUF1+3 DEVTST:      IOT
                     BPL DUNTST               .WORD DEVTST
                                              .BYTE WAITR,SLOT2

                                              IOT
                                              .WORD SLOT2DEV
                                              .BYTE INIT, SLOT4
```

Programs A and B do two successive reads from the same device into two different buffers. Since the devices are the same, IOX waits for the first read to finish before allowing the second to begin.

In Program A, we wish to process buffer 1. To have issued a Waitr for the device associated with slot 2 could have meant waiting also for the device at slot 3 if that device were in conflict. Hence, testing the Done bit in the buffer header is the proper choice.

In Program B, we wish control of the device at slot 2, so that it can be assigned to another slot and so we must know its availability. Therefore, Waitr is appropriate.

7.5.6 Single Buffer Transfer on One Device

```
    A:  IOT                       ;TRAP TO IOX
        .WORD BUF1                ;SPECIFY BUFFER
        .BYTE READ,SLOT3          ;READ FROM DEVICE AT
                                  ;SLOT 3 INTO BUFFER

 BUSY:  IOT                       ;TRAP TO IOX
        .WORD BUSY                ;SPECIFY BUSY RETURN ADDRESS
        .BYTE WAITR,SLOT3         ;WAIT FOR DEVICE AT SLOT
                                  ;3 TO FINISH READING
        (process buffer 1)

        JMP A
```

The program segment above includes a Waitr which goes to a Busy Return address that is its own IOT -- continuously testing the device at slot 3 for availability. In this instance, involving only a single device and a single buffer, a Done condition in the Buffer 1 Status byte can be inferred from the availability of the device at slot 3. This knowledge assures us that all data requested for Buffer 1 is available for processing.

Testing the Done Bit of Buffer 1 might have been used instead, but was not necessary with only one device operating. Moreover, a Waitr, unlike a Done Bit test, would detect a timeout on the LSR if that device happened to be associated with slot 3.

7.5.7 Double Buffering

```
        IOT                      ;TRAP TO IOX
        .WORD BUF1               ;SPECIFY BUFFER 1
        .BYTE READ,SLOT3         ;READ FROM DEVICE AT
                                 ;SLOT 3 INTO BUFFER 1

    A:  IOT                      ;TRAP TO IOX
        .WORD BUF2               ;SPECIFY BUFFER 2
        .BYTE READ,SLOT3         READ FROM DEVICE AT SLOT
                                 ;3 INTO BUFFER 2

        (process BUF1 concurrent with Read into BUF2)

    B:  IOT                      ;TRAP TO IOX
        .WORD BUF1               ;SPECIFY BUFFER 1
        .BYTE READ,SLOT3         ;READ FROM DEVICE AT
                                 ;SLOT 3 INTO BUFFER 1

        (process BUF2 concurrent with Read into BUF1)

        JMP A
```

The example above illustrates a time-saving double-buffer scheme whereby data is processed in Buffer 1 at the same time as new data is being read into Buffer 2; and, sequentially, data is processed in Buffer 2 at the same time as new data is being read into Buffer 1.

Because IOX ensures that the requested device is free before initiating the command, the subsequent return of control from the IOT at A implies that the read prior to A is complete; that is, that buffer 1 is available for processing. Similarly, the return of control from the IOT at B implies that buffer 2 is available. Waitr's are not required because IOX has automatically ensured the device's availability before initiating each Read.

7.5.8 Readr (Real-time Read)

```
IOT
.WORD (address of first word of the buffer header)
.BYTE 13, (slot number)
.WORD (done-address)
```

The Readr command functions as the Read except that upon completion of the data transfer, program control goes to the specified Done-address at the priority level of the device. Readr is used when you wish to execute a segment of your program immediately upon completing the data transfer. IOX goes to the Done address by executing a JSR R7, Done-address.

The general registers, which were saved when the last character interrupt occurred, are on the SP stack in the order indicated below:

```
(SP)→      Return address to IOX
           R5
           R4
           R3
           R2
           R1
           R0
```

Return to IOX is accomplished by an RTS R7 instruction. IOX will then restore all registers and return to the interrupted program. Care should be taken in initiating another data transfer if the specified device can conflict with device requests at other priority levels. Waitr cannot be used to resolve conflict situations between priority levels.

7.5.9 Writr (Real-time Write)

```
IOT
.WORD (address of first word of the buffer header)
.BYTE 14, (slot number of device)
.WORD (done address)
```

The Writr command functions as the Write except that, upon completion of the data transfer, program control goes to the specified Done-address at the priority level of the device. IOX goes to the Done-address by executing a JSR R7, Done-address. The condition of the general registers and the return to IOX are the same as for Readr. Writr is used when you wish to execute a segment of your program immediately upon completing the data transfer.

As in the Readr, care should be taken in initiating another data transfer if the specified device can conflict with device requests at the priority level of the calling program.

7.6 REENABLING THE READER AND RESTARTING

7.6.1 Seek

```
IOT
.WORD 0
.BYTE 5, (slot number of LSR or HSR)
```

The Seek command clears IOX's internal End-Of-Medium (EOM) indicator on the LSR or HSR, making possible a subsequent read on those devices. With no EOM, an EOF cannot occur. The device associated with the specified slot remains Inited.

7.6.2 Restart

```
IOT
.WORD (address to restart)
.BYTE 3,0
```

This command designates an address at which to restart your program. After this command has been issued, typing CTRL/P on the KBD will transfer program control to the restart address, providing there is no LSR read in progress. In such a case, the LSR must be turned off (causing a timeout) before typing a CTRL/P. If the Restart address is designated as 0, the CTRL/P Restart capability is disabled.

The Restart command does not cancel any I/O in progress. It is the program's responsibility in its restart routine to clean up any I/O by executing a RESET command and ensuring that the stack pointer is reset.

7.7 FATAL ERRORS

Fatal errors result in program termination and a jump to location 40_8 (loaded with a HALT by IOX), with R0 set to the error code and R1 set as follows:

If the fatal error was due to an illegal memory reference (code 0), R1 will contain the PC at the time of the error.

If the fatal error was due to an error coded in the range 1-5, R1 will point to some element in the IOT argument list or to the instruction following the argument list, depending on whether IOX has finished decoding the arguments when it detects the error.

Fatal Error Code	Reason
0	Illegal Memory Reference, SP overflow, illegal instruction
1	Illegal IOX command
2	Slot out of range
3	Device out of range
4	Slot not inited
5	Illegal data mode

Note that the SP stack contains the value of the registers at the time of the error, namely

```
(SP)→ R5
      R4
      R3
      R2
      R1
      R0
      PC
      Processor Status (PS)
```

(See Section 7.3.3.1 for a discussion of non-fatal errors.)

7.8 EXAMPLE OF PROGRAM USING IOX

This program is used to duplicate paper tape. Note that it could be altered by changing the device code at RDEV or PDEV. For instance, the program could easily be made to list a tape.

```
R0=%0
R1=%1
R2=%2
R3=%3
R4=%4
R6=%6
KSLOT=0
TSLOT=1
RSLOT=3
PSLOT=4
RESET=2
RESTRT=3
INIT=1
```

```
WAITR=4
READ=11
WRITE=12
EOF=20000
CR=15                           ;CR ASSIGNED ASCII CODE FOR CARRIAGE RETURN
LF=12                           ;LF ASSIGNED ASCII CODE FOR LINE FEED

        .=1000
MSG1:   0                       ;CANNED MESSAGE
        0                       ;FORMATTED ASCII
MSG1BC: END1-MSG1BC-2           ;BYTE COUNT
        .BYTE   CR,LF
        .ASCII  / PLACE TAPE IN READER/
        .BYTE   CR,LF
        .ASCII  / STRIKE CR WHEN READY/
END1:   .EVEN
BUF3:   2                       ;BUFFER SIZE
        0                       ;FORMATTED ASCII MODE
        0                       ;BC
        0                       ;CR LF

RDEV:   5                       ;DEVICE CODE FOR HSR
PDEV:   6                       ;DEVICE CODE FOR HSP

BUF1:   100                     ;BUFFER SIZE
        3                       ;CODE FOR UNFORMATTED BINARY
        100                     ;SPECIFIES NUMBER OF BYTES FOR TRANSFER
        .=.+100                 ;RESERVES STORAGE FOR DATA
BUF2:   100                     ;BUFFER SIZE
        3                       ;CODE FOR UNFORMATTED BINARY
        100                     ;SPECIFIES NUMBER OF BYTES FOR TRANSFER
        .=.+100                 ;RESERVES STORAGE FOR DATA
BEGIN:  MOV     #500,R6         ;SPECIFY ADDRESS FOR BOTTOM OF STACK

        IOT
        0
        .BYTE   RESET,0         ;INITIALIZATION

        IOT
        BEGIN                   ;'BEGIN' SPECIFIED AS RESTART
        .BYTE   RESTRT,0        ;ADDRESS FOR CTRL P
        MOV     #100,BUF1+4     ;SET UP INITIAL BC ON BUF1
        MOV     #100,BUF2+4     ;SET UP INITIAL BC ON BUF2

        IOT                     ;TYPE OUT DIRECTIONS
        MSG1
        .BYTE WRITE,TSLOT

        IOT                     ;READ A CR,LF
        BUF3
        .BYTE READ,KSLOT

A:      IOT                     ;WAIT FOR HIM TO TYPE A CARRIAGE RETURN,
                                ;LINE FEED
        A
        .BYTE WAITR,KSLOT

        IOT                     ;INIT READER
        RDEV
        .BYTE INIT,RSLOT

        IOT                     ;INIT PUNCH
        PDEV
        .BYTE INIT,PSLOT
        IOT                           ;START FIRST READ
        BUF1
        .BYTE READ,RSLOT

LOOP:   IOT                           ;READ INTO 2ND BUFFER
        BUF2
        .BYTE READ,RSLOT

        BIT     #EOF BUF1+2           ;END OF FILE?
        BNE     BEGIN                 ;YES
                                      ;NO

        IOT                           ;WRITE OUT THIS BUFFER
        BUF1
        .BYTE WRITE,PSLOT

C:      IOT                           ;WAIT TILL DEVICE HAS FINISHED
        C
        .BYTE WAITR,PSLOT

        IOT                           ;READ INTO 1ST BUFFER
        BUF1
        .BYTE READ,RSLOT

        BIT     #EOF,BUF2+2           ;END OF FILE?
        BNE     BEGIN

        IOT                           ;WRITE OUT BUFFER 2
        BUF2
        .BYTE WRITE,PSLOT

B:      IOT                           ;WAIT TILL DEVICE HAS FINISHED
        B
        .BYTE WAITR,PSLOT
        BR      LOOP
        .END    BEGIN
```

7.9 IOX INTERNAL INFORMATION

7.9.1 Conflict Byte/Word

The IOX Conflict byte (in IOXLPT, Conflict Word) contains the status (busy or free) of all devices as well as whether or not an echo is in progress. Bit 0 is the echo bit, bits 1-6 (and 8 in IOXLPT) refer to the corresponding codes for devices:

			If Bit is Set
Bit	0	=	Echo in progress
Bit Device	1	=	KBD busy
Bit Device	2	=	TTY busy
Bit Device	3	=	LSR busy

Bit Device	4	=	LSP busy
Bit Device	5	=	HSR busy
Bit Device	6	=	HSP busy
Bit Device	8 10_8	=	LPT busy

In IOXLPT, the Conflict Byte is expanded to a word in order to accommodate the line printer, there being no bit 8 to correspond with that device's code of 10 (octal) (the lowest available code for an output device - see Section 7.9.5.1).

Device	All Possible Conflicting Devices	Conflict Number
KBD	Echo, KBD, TTY, LSR, LSP	37
TTY	Echo, KBD, TTY, LSP	27
LSR	KBD, LSR	12
LSP	Echo, KBD, TTY, LSP	27
HSR	HSR	40
HSP	HSP	100
LPT	LPT	400

For each of the devices in the left hand column, all the possible conflicts are listed along with their respective conflict numbers. These numbers, representing bit patterns of the devices listed in column two above, are used to resolve any conflicting requests for devices. The appropriate number is masked with the conflict byte. If the result is zero, there are no conflicts and the device being tested has its bit set allowing data transfer to begin.

7.9.2 Device Interrupt Table (DIT)

Each device interrupt handler has associated with it a Device Interrupt Table (DIT) containing information that the handler needs:

DIT	Checksum
DIT+2	Byte size from buffer header
DIT+4	Address of Mode byte in buffer header
DIT+6	Byte Location Pointer
DIT+10	Byte Count
DIT+12	Device code

DIT+14	Real time done-address
DIT+16	Address of device's data buffer register

The device interrupt routines gain access to the proper data by means of the DIT entry. When a transfer is complete, they set the appropriate bits in the buffer header pointed to by the DIT contents.

7.9.3 Device Status Table (DST)

The Device Status Table (DST) is used by IOX to check for EOF conditions. This table contains a word for each device indicating an EOM condition with a 1. When an EOM condition is recognized on input, IOX not only sets the appropriate bit in the buffer status byte associated with the data transfer, it also records this occurrence in the DST. When a data transfer command is given, IOX checks the DST for the EOM condition. If the appropriate word has a value of 1, IOX sets EOF in the Status byte of the current-command buffer. Since EOF is only possible for the LSR (code 3), and HSR (code 5), the words corresponding to those devices are the only ones that can ever be set to 1.

7.9.4 Teletype Hardware Tab Facility

If the Teletype model has a hardware tab facility, teleprinter output can be speeded up by:

1. For IOX, deleting the code from I.TTYCK+6 through I.TAB3+3.

2. For IOXLPT, skipping the code from I.IOLF through I.TAB3+3 (for the teleprinter only - not the line printer).

7.9.5 Adding Devices To IOX

In order to add a device to IOX the following tasks must be done:

1. Assign a legal code to the device

2. Modify the IOX tables

3. Provide an interrupt routine to handle data for the device.

The line printer (in IOXLPT) will be used as an example throughout this discussion.

7.9.5.1 Device Codes - The numbers from 7 to 17 (octal) are available for new-device codes, with the exception of 10 (octal) in the IOXLPT version. This code has been assigned to the line printer. The device code must be odd for an input device and even for an output device. This is so a check can be made for command/device correspondence; i.e., for a Read from an input device or a Write to an output device.

If the newest device was assigned a number that is higher than the codes of all the other devices, I.MAXDEV must be redefined to that value. This is so an out-of-range device specification in an Init command can be detected. In IOXLPT, I.MAXDEV=10.

Since each device code functions as an index in several word tables, the entries relating to a given device must be placed at the same relative position in each appropriate table. That is, the code number must indicate how many words into the table the entry for that device will be found. This, of course, means accounting for any unused space preceding the entry, if the codes are not assigned in strict sequence. Table entries for the line printer are found at the 10th (octal) word past the table tag, i.e., at Table+20.

7.9.5.2 Table Modification -

1. I.FUNC - Each entry is the octal value of the bit pattern in the device Control/Status Register that enables the corresponding device and/or any interrupt facility it has. Bit setting this number into the device's Control/Status register turns the device on; bit clearing turns it off. Determine this value for the device to be added, and place the entry in the appropriate device position in the table. For example, the line printer Control/Status Register has an Interrupt Enable facility in bit 6. This pattern of 100 is the LPT entry, and is located at I.FUNC+20.

2. I.SCRTAB - This table contains the addresses of the device Control/Status registers. The line printer entry I.LPTSCR has the value 177514, and is located at I.SCRTAB+20.

3. I.DST - (Refer to Section 7.9.3.) Create an entry of 0 for the device in the proper table location. Inserting a word of 0 at I.DST+20 created a device status entry for the line printer.

4. I.CONSIT - An entry in this table is used to set or clear a device's busy/free bit in the Conflict Byte (Conflict Word in IOXLPT). (See Section 7.9.1, and 5. below.) Each value is obtained by setting one bit only - the bit number corresponding to the device number. The line printer, being device 10(octal), has a value of 400(octal) (bit 10 set) and is located at I.CONSIT+20.

   In the IOX version without the line printer, entries to this table are found in the high-order bytes of Table I.CONFLC. One more input device entry can be added to it. In IOXLPT, however, I.CONSIT is a separate word table, allowing eight more devices (four input and four output) to be added. Byte operations in the IOX I.CONSIT became word operations in IOXLPT to adapt to this expansion.

5. I.CONFLC - (Refer to Section 7.9.1 on Conflict Byte/Word.) Entries are bit patterns of conflicting devices. Since the line printer can only conflict with itself, the I.CONFLC entry is equal to the I.CONSIT entry. As in the I.CONSIT table, byte operations were changed to word operations for I.CONFLC in IOXLPT.

6. Create a DIT for the device (refer to Section 7.9.2) by assigning a DIT label and seven words of 0. If it is an output device, the address of the Device Buffer Register must be added as an eighth word.

7. I.INTAB - This is a table of DIT addresses. Place the label of the DIT (mentioned in 6. above) in the correct position in the table. I.INTAB+20 contains the line printer entry I.LPTDIT.

7.9.5.3 Interrupt Routines - Write (and assign a label to) an interrupt routine for the device to:

1. Get a character
2. Check for errors by means of the device Control/Status register
3. Do character interpretation according to the device and mode
4. Get a character in or out of the buffer
5. Update IOX's Byte Count
6. Compare IOX's Byte Count to User's Byte Count and Buffer size specification
7. Return for next character

Place the label of the interrupt routine at the address of the device vector, and follow it with the value of the interrupt priority in bits 7, 6, and 5. I.LPTIR, the address of the line printer interrupt routine, is at location 200. Location 202 contains the value 200 (indicating priority level 4).

If the device to be added is similar to the other single-character devices, steps 3-7 above can be performed by IOX as indicated below:

There are two routines, I.INPUT and I.OUTPUT, that are called from the interrupt routines. These routines mainly perform common functions for input and output devices. They are called as follows:

```
        JSR  R5,I.INPUT     and     JSR  R5,I.OUTPUT
```

At the location following one of these calls is the DIT for the proper device. The routine is thus able to use R5 to reference the DIT entries.

I.INPUT and I.OUTPUT also contain device-dependent code to perform functions such as tab counters for the teleprinter and line printer, and deletion of carriage returns in Formatted ASCII mode for the line printer. The device index value is used to identify the device. For the line printer, a symbol I.LPT, has been assigned the value 20 for convenient reference to the device index.

CHAPTER 8

FLOATING POINT MATH PACKAGE OVERVIEW[1]

The new Floating-Point Math Package, FPMP-11, is designed to bring the 2/4 word floating point format of the FORTRAN environment to the paper tape software system of the PDP-11. The numerical routines in FPMP-11 are the same as those of the DOS/BATCH FORTRAN Operating Time System (OTS). TRAP and error handlers have been included to aid in interfacing with the FORTRAN routines.

FPMP-11 provides an easy means of performing basic arithmetic operations such as add, subtract, multiply, divide, and compare. It also provides transcendental functions (SIN, COS, etc.), type conversions (integer to floating-point, 2-word to 4-word, etc.), and ASCII conversions (ASCII to 2-word floating-point, etc.).

Floating-point notation is particularly useful for computations involving numerous multiply and divide operations where operand magnitudes may vary widely. FPMP-11 stores very large and very small numbers by saving only the significant digits and computing an exponent to account for leading and trailing zeros.

To conserve core space in a small system, FPMP-11 can be tailored to include only those routines needed to run a particular user program.

For more information on FPMP-11, refer to the FPMP-11 User's Manual (DEC-11-NFPMA-A-D) and to Appendix H of this manual.

[1] FPMP is not currently available for the LSI-11 (PDP-11/03).

CHAPTER 9

PROGRAMMING TECHNIQUES

This chapter presents various programming techniques. They can be used to enhance your programming and to make optimum use of the PDP-11 processor. The reader is expected to be familiar with the PAL-11 assembly language (Chapters 1 & 2).

9.1 WRITING POSITION INDEPENDENT CODE

When a standard program is available for different users, it often becomes useful to be able to load the program into different areas of core and to run it there. There are several ways to do this:

1. Reassemble the program at the desired location.
2. Use a relocating loader which accepts specially coded binary from the assembler.
3. Have the program relocate itself after it is loaded.
4. Write code which is position independent.

On small machines, reassembly is often performed. When the required core is available, a relocating loader (usually called a linking loader) is preferable. It generally is not economical to have a program relocate itself since hundreds or thousands of addresses may need adjustment. Writing position independent code is usually not possible because of the structure of the addressing of the object machine. However, on the PDP-11, position independent code (PIC) is possible.

PIC is achieved on the PDP-11 by using addressing modes which form an effective memory address relative to the Program Counter (PC). Thus, if an instruction and its object(s) are moved in such a way that the relative distance between them is not altered, the same offset relative to the PC can be used in all positions in memory. Thus, PIC usually references locations relative to the current location. PIC may make absolute references as long as the locations referenced stay in the same place while the PIC is relocated. For example, references to interrupt and trap vectors are absolute, as are references to device registers in the external page and direct references to the general registers.

9.1.1 Position Independent Modes

There are three position independent modes or forms of instructions. They are:

1. Branches -- the conditional branches, as well as the unconditional branch, BR, are position independent since the branch address is computed as an offset to the PC.

2. Relative Memory References -- any relative memory reference of the form

   ```
   CLR X
   MOV X,Y
   JMP X
   ```

   is position independent because the assembler assembles it as an offset indexed by the PC. The offset is the difference between the referenced location and the PC. For example, assume the instruction CLR 200 is at address 100:

   ```
   100/  005067        ;FIRST WORD OF CLR 200
   102/  000074        ;OFFSET = 200-104
   ```

   The offset is added to the PC. The PC contains 104, i.e., the address of the word following the offset.

   Although the form CLR X is position independent, the form CLR @X is not. Consider the following:

   ```
   S:  CLR @X        ;CLEAR LOCATION A
        .
        .
        .
   X:  .WORD A       ;POINTER TO A
        .
        .
        .
   A:  .WORD 0
   ```

   The contents of location X are used as the address of the operand in the location labeled A. Thus, if all of the code is relocated, the contents of location X must be altered to reflect the new address of A. If A, however, was the name associated with some fixed location (e.g., trap vector, device register), then statements S and X would be relocated and A would remain fixed. Thus, the following code is position independent.

   ```
       A = 36        ;ADDRESS OF SECOND WORD OF
                     ;TRAP VECTOR
   S:  CLR @X        ;CLEAR LOCATION A
        .
        .
        .
   X:  .WORD A       ;POINTER TO A
   ```

3. Immediate Operands -- The assembler addressing form #X specifies immediate data, that is, the operand is in the instruction. Immediate data is position independent since it is a part of the instruction and is moved with the

instruction. Immediate data is fetched using the PC in the autoincrement mode.

As with direct memory references, the addressing form @#X is not position independent. As before, the final effective address is absolute and points to a fixed location not relative to the PC.

9.1.2 Absolute Modes

Any time a memory location or register is used as a pointer to data, the reference is absolute. If the referenced data is fixed in memory, independent of the position of the PIC (e.g., trap-interrupt vectors, device registers), the absolute modes must be used.[1] If the data is relative to the PIC, the absolute modes must not be used unless the pointers involved are modified. The absolute modes are:

@X		Location X is a pointer
@#X		The immediate word is a pointer
(R)		The register is a pointer
(R)+	and -(R)	The register is a pointer
@(R)+	and @-(R)	The register points to a pointer
X(R)	R≠6 or 7	The base, X, modified by (R) is the address of the operand
@X(R)		The base, modified by (R), is a pointer

The non-deferred index modes and stack operations require a little clarification. As described in Sections 3.6.10 and 9.1.1, the form X(7) is the normal mode to reference memory and is a relative mode. Index mode, using a stack pointer (SP or other register) is also a relative mode and may be used conveniently in PIC. Basically, the stack pointer points to a dynamic storage area and index mode is used to access data relative to the pointer. The stack pointer may be initially set up by a position independent program as shown in Section 9.1.4.1. In any case, once the pointer is set up, all data on the stack is referenced relative to the pointer. It should also be noted that since the form 0 (SP) is considered a relative mode so is its equivalent @SP. In addition, the forms (SP)+ and -(SP) are required for stack pops and pushes.

9.1.3 Writing Automatic PIC

Automatic PIC is code which requires no alteration of addresses or pointers. Thus, memory references are limited to relative modes unless the location referenced is fixed (trap-interrupt vectors, etc.). In addition to the above rules, the following must be observed:

1. Start the program with .=0 to allow easy relocation using the Absolute Loader (see Chapter 6).

[1] When PIC is not being written, references to fixed locations may be performed with either the absolute or relative forms.

2. All location setting statements must be of the form .=.±X or .= function of tags within the PIC. For example, .=A+10 where A is a local label.

3. There must not be any absolute location setting statements. This means that a block of PIC cannot set up trap and/or interrupt vectors at load time with statements such as:

```
.=34
.WORD  TRAPH,340   ;TRAP VECTOR
```

   The Absolute Loader, when it is relocating PIC, relocates all data by the load bias (see Chapter 6). Thus, the data for the vector would be relocated to some other place. Vectors may be set at execution time (see Section 9.1.4).

9.1.4 Writing Non-Automatic PIC

Often it is not possible or economical to write totally automated PIC. In these cases, some relocation may be easily peformed at execution time. Some of the required methods of solution are presented below. Basically, the methods operate by examining the PC to determine where the PIC is actually located. Then a relocation factor can be easily computed. In all examples, it is assumed that the code is assembled at zero and has been relocated somewhere else by the Absolute Loader.

9.1.4.1 Setting Up The Stack Pointer - Often the first task of a program is to set the stack pointer (SP). This may be done as follows:

```
          .=0              ;BEG IS THE FIRST INSTRUCTION OF
                           ;THE PROGRAM
BEG:      MOV   PC,SP      ;SP=ADR BEG+2
          TST   -(SP)      ;DECREMENT SP BY 2.
                           ;A PUSH ONTO THE STACK WILL STORE
                           ;THE DATA AT BEG-2.
```

9.1.4.2 Setting Up A Trap or Interrupt Vector - Assume the first word of the vector is to point to location INT which is in PIC.

```
X:   MOV PC,R0          ;R0 = ADR X+2
     ADD #INT-X-2,R0    ;ADD OFFSET
     MOV R0,@#VECT      ;MOVE POINTER TO VECTOR
```

The offset INT-X-2 is equivalent to INT-(X+2) and X+2 is the value of the PC moved by statement X. If PC is the PC that was assumed for the program when loaded at 0, and if PC is the current real PC, then the calculation is:

$$INT-PC_o+PC_n=INT+(PC_n-PC_o)$$

Thus, the relocation factor, PC_n-PC_o, is added to the assembled value of INT to produce the relocated value of INT.

9.1.4.3 Relocating Pointers - If pointers must be used, they may be relocated as shown above. For example, assume a list of data is to be accessed with the instruction

```
        ADD (R0)+,R1
```

The pointer to the list, list L, may be calculated at execution time as follows:

```
        M:          MOV   PC,R0        ;GET CURRENT PC
                    ADD   #L-M-2,R0    ;ADD OFFSET
```

Another variation is to gather all pointers into a table. The relocation factor may be calculated once and then applied to all pointers in the table in a loop.

```
        X:          MOV   PC,R0        ;RELOCATE ALL ENTRIES IN PTRTBL
                    SUB   #X+2,R0      ;CALCULATE RELOCATION FACTOR
                    MOV   #PTRTBL,R1   ;GET AND RELOCATE A POINTER
                    ADD   R0,R1        ;  TO PTRTBL
                    MOV   #TBLLEN,R2   ;GET LENGTH OF TABLE
     LOOP:          ADD   R0,(R1)+     ;RELOCATE AN ENTRY
                    DEC   R2           ;COUNT
                    BGE   LOOP         ;BRANCH IF NOT DONE
```

Care must be exercised when restarting a program which relocates a table of pointers. The restart procedure must not include the relocating again, i.e., the table must be relocated exactly once after each load.

9.2 LOADING UNUSED TRAP VECTORS

One of the features of the PDP-11 is the ability to trap on various conditions such as illegal instructions, reserved instructions, power failure, etc. However, if the trap vectors are not loaded with meaningful information, the occurrence of any of these traps will cause unpredictable results. By loading the vectors as indicated below, it is possible to avoid these problems as well as gain meaningful information about any unexpected traps that occur. This technique, which makes it easy to identify the source of a trap, is to load each unused trap vector with:

```
                    .=trap address
                    .WORD .+2,HALT
```

This will load the first word of the vector with the address of the second word of the vector (which contains a HALT). Thus, for example, a halt at location 6 means that a trap through the vector at location 4 has occurred. The old PC and status may be examined by looking at the stack pointed to by register 6.

The trap vectors of interest are:

Vector Location	Halt At Location	Meaning
4	6	Bus Error; Illegal Instruction; Stack Overflow; Nonexistent Memory; Nonexistent Device; Word Referenced at Odd Address
10	12	Reserved Instruction

14	16	Trace Trap Instruction (000003) or T-bit Set in Status Word (used by ODT)
20	22	IOT Executed (used by IOX)
24	26	Power Failure or Restoration
30	32	EMT Executed (used by FPP-11)
34	36	TRAP Executed

9.3 CODING TECHNIQUES

Because of the great flexibility in PDP-11 coding, time- and space-saving ways of performing operations may not be immediately apparent. Some comparisons follow.

9.3.1 Altering Register Contents

The techniques described in this section take advantage of the automatic stepping feature of autoincrement and autodecrement modes when used especially in TST and CMP instructions. These instructions do not alter operands. However, it is important to make note of the following:

- These alternative ways of altering register contents affect the condition codes differently.
- Register contents must be even when stepping by 2.
- These techniques work properly only if the registers are pointing to an existing memory location; otherwise, a trap is generated.

1. Adding 2 to a register might be accomplished by ADD #2,R0. However, this takes two words, whereas TST (R0)+ which also adds 2 to a register, takes only one word.

2. Subtracting 2 from a register can be done by the complementary instructions SUB #2,R0 or TST -(R0) with the same conditions as in adding 2.

3. This can be extended to adding or subtracting 2 from two different registers, or 4 from the same register, in one single-word instruction:

```
CMP (R0)+,(R0)+          ;ADD 4 TO R0
CMP -(R1),-(R1)          ;SUBTRACT 4 FROM R1
CMP (R0)+,-(R1)          ;ADD 2 TO R0, SUBTRACT 2 FROM R1
CMP -(R3),-(R1)          ;SUBTRACT 2 FROM BOTH R3 AND R1
CMP (R3)+,(R0)+          ;ADD 2 TO BOTH R3 AND R0
```

4. Variations of the examples above can be employed if the instructions operate on bytes and one of the registers is the Stack Pointer. These examples depend on the fact that the Stack Pointer (as well as the PC) is always autoincremented or autodecremented by 2, whereas registers R0-R5 step by 1 in byte instructions.

```
        CMPB (SP)+,(R3)+          ;ADD 2 TO SP AND 1 TO R3
        CMPB -(R3),-(SP)          ;SUBTRACT 1 FROM R3 AND 2 FROM SP
        CMPB (R3)+,-(SP)          ;ADD 1 TO R3, SUBTRACT 2 FROM SP
```

5. Popping an unwanted word off the processor stack (adding 2 to register 6) and testing another value can be two separate instructions or one combined instruction:

```
        TST (SP)+                 ;POP WORD
        TST COUNT                 ;SET CONDITION CODES FOR COUNT
or
        MOV COUNT,(SP)+           ;POP WORD & SET CODES FOR COUNT
```

The differences are that the TST instructions take three words and clear the Carry bit, and the MOV instruction takes two words and doesn't affect the Carry bit.

9.3.2 Subroutines

1. Condition codes set within a subroutine can be used to conditionally branch upon return to the calling program, since the RTS instruction does not affect condition codes.

```
              JSR PC,X            ;CALL SUBROUTINE X
              BNE ABC             ;BRANCH ON CONDITION SET
                 .                ;IN SUBROUTINE X
                 .
                 .
       X:                         ;SUBROUTINE ENTRY
                 .
                 .
                 .
              CMP R2,DEF          ;TEST CONDITION
              RTS PC              ;RETURN TO CALLING PROGRAM
```

2. When a JSR first operand register is not the PC, data stored following a subroutine call can be accessed within the subroutine by referencing the register. (The register contains the return address.)

```
              JSR R5,Y
              .WORD HIGH
              .WORD LOW
                 .                ;LATEST R5 VALUE WILL POINT HERE
                 .
                 .
       Y:     MOV (R5)+,R2        ;VALUE OF HIGH ACCESSED
              MOV (R5)+,R4        ;VALUE OF LOW ACCESSED
                 .
                 .
                 .
              RTS R5              ;RETURN TO LOCATION
                                  ;CONTAINED IN R5
```

Another possibility is:

```
              JSR R5,SUB
              BR PSTARG          ;LOW-ORDER BYTE IS OFFSET TO RETURN
                                 ;ADDRESS, WHICH EQUALS NO. OF ARGS.
              .WORD A            ;ADDRESS OF ARG A
              .WORD B            ;ADDRESS OF ARG B
              .WORD C            ;ADDRESS OF ARG C
                  .
                  .
                  .
PSTARG:                          ;RETURN ADDRESS
                  .
                  .
                  .
    SUB:      MOVB@R5,COUNT      ;GET NO. OF ARGS FROM LOW BYTE
                                 ;OF BR (IF DESIRED).
              MOV @14(R5),R2     ;E.G., GET 6TH ARGUMENT
              MOV @6(R5),R1      ;GET 3RD ARGUMENT
                  .
                  .
                  .
              RTS R5             ;RETURNS TO BRANCH WHICH JUMPS PAST
                                 ;ARG LIST TO REAL RETURN ADDRESS
```

In the example above, the branch instruction contributes two main advantages:

1. If R5 is unaltered when the RTS is executed, return will always be to the branch instruction. This ensures a return to the proper location even if the length of the argument list is shorter or longer than expected.

2. The operand of the branch, being an offset past the argument list, provides the number of arguments in the list.

Arguments can be made sharable by separating the data from the main code. This is easily accomplished by treating the JSR and its return as a subroutine itself:

```
    CALL:     .                  ARGLST:    JSR R5,SUB
              .                             BR PSTARG
              .
            JSR PC,ARGLST                   .WORD A
              .                               .
              .                               .
              .                               .
```

3. The examples above all demonstrate the calling of subroutines from a non-reentrant program. The called subroutine can be either reentrant or non-reentrant in each case. The following example illustrates a method of also allowing calling programs to be reentrant. The arguments and linkage are first placed on the stack, simulating a JSR R5,SUB, so that arguments are accessed from the subroutine via X(R5). Return to the calling program is executed from the stack.

```
CALL:
              .
              .
              .
         MOV R5,-(SP)      ;SAVE R5 ON STACK.
         MOV JSBR,-(SP)    ;PUSH INSTRUCTION JSR R6,@R5 ON
              .            ;STACK.  PUSH ADDRESSES OF ARGU-
              .            ;MENTS ON STACK IN REVERSE ORDER
              .            ;(SEE BELOW).
         MOV BRN,-(SP)     ;PUSH BRANCH INSTRUCTION ON STACK
    X:   MOV SP,R5         ;MOVE ADDRESS OF BRANCH TO R5.
         JSR PC,SUB        ;CALL SUB AND SAVE RETURN ON STACK.
 RET:    MOV (SP)+,R5      ;RESTORE OLD R5 UPON RETURN.
              .
              .            ;DATA AREA OF PROGRAM.
              .
JSBR:    JSR R6,@R5
 BRN:    BR .+N+N+2        ;BRANCH PAST N WORD ARGUMENTS
```

The address of an argument can be pushed on the stack in several ways. Three are shown below.

a. The arguments A, B, and C are read-only constants which are in memory (not on the stack):

```
MOV #C,-(SP)     ;PUSH ADDRESS OF C
MOV #B,-(SP)     ;PUSH ADDRESS OF B
MOV #A,-(SP)     ;PUSH ADDRESS OF A
```

b. Arguments A, B, and C have their addresses on the stack at the Lth, Mth, and Nth bytes from the top of the stack.

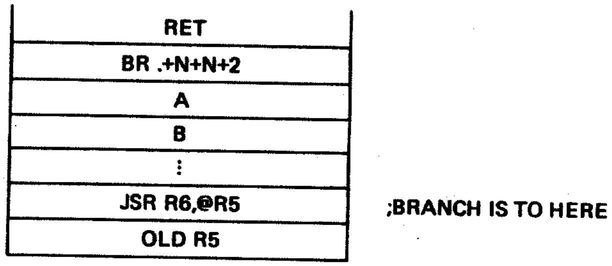
```
MOV N(SP),-(SP)      ;PUSH ADDRESS OF C
MOV M+2(SP),-(SP)    ;PUSH ADDRESS OF B
MOV L+4(SP),-(SP)    ;PUSH ADDRESS OF A
```

Note that the displacements from the top of the stack are adjusted by two for each previous push because the top of the stack is being moved on each push.

c. Arguments A, B, and C are on the stack at the Lth, Mth, and Nth bytes from the top but their addresses are not.

```
MOV #N+2,-(SP)   ;PUSH DISPLACEMENT TO ARGUMENT
ADD SP,@SP       ;CALCULATE ACTUAL ADDRESS OF C
MOV #M+4,-(SP)
ADD SP,@SP       ;ADDRESS OF B
MOV #L+6,-(SP)
ADD SP,@SP       ;ADDRESS OF A
```

When subroutine SUB is entered, the stack appears as follows:



Subroutine SUB returns by means of an RTS R5, which places R5 into the PC and pops the return address from the stack into R5. This causes the execution of the branch because R5 has been loaded (at location X) with the address of the branch. The JSR branched to then returns control to the calling program, and in so doing, moves the current PC value into the SP, thereby removing everything above the old R5 from the stack. Upon return at RET, this too is popped, restoring the original R5 and SP values.

4. The next example is a recursive subroutine (one that calls itself). Its function is to look for a matching right parenthesis for every left parenthesis encountered. The subroutine is called by JSR PC,A whenever a left parenthesis is encountered (R2 points to the character following it). When a right parenthesis is found, an RTS PC is executed, and if the right parenthesis is not the last legal one, another is searched for. When the final matching parenthesis is found, the RTS returns control to the main program.

```
A:     MOVB (R2)+,R0      ;GET SUCCESSIVE CHARACTERS.
       CMPB #'(,R0        ;LOOK FOR LEFT PARENTHESIS.
       BNE B              ;FOUND?
       JSR PC,A           ;LEFT PAREN FOUND, CALL SEL.
       BR A               ;GO LOOK AT NEXT CHARACTER

B:     CMPB #'),R0        ;LEFT PAREN NOT FOUND, LOOK FOR
                          ;RIGHT PAREN.
       BNE A              ;FOUND? IF NOT, GO TO A.
       RTS PC             ;RETURN PAREN FOUND. IF NOT LAST,
                          ;GO TO B. IF LAST, GO TO MAIN PROGRAM.
```

5. The example below illustrates the use of co-routines, called by JSR PC,@(SP)+. The program uses double buffering on both input and output, performing as follows:

```
Write O1   )                      Write O2   )
Read I1    }  concurrently        Read I2    }  concurrently
Process I2 )                      Process I1 )
```

JSR PC,@(SP)+ always performs a jump to the address specified on top of the stack and replaces that address with the new return address. Each time the JSR at B is executed, it jumps to a different location; initially to A and thereafter to the location following the JSR executed prior to the one at B. All other JSR's jump to B+2.

```
          PC=%7
BEGIN:    (do I/O resets, inits, etc.)
            .
            .
            .
          IOT                ;READ INTO I1 TO START PROCESS
          .WORD I1
          .BYTE READ,INSLOT
          MOV #A,-(6)        ;INITIALIZE STACK FOR FIRST JSR
B:        JSR PC,@(6)+       ;DO I/O FOR O1 AND I1 OR O2 AND I2
            .
            .      perform processing
            .
          BR B               ;MORE I/O
                             ;END OF MAIN LOOP
                             ;I/O CO-ROUTINES
A:        IOT                ;READ INTO I2
          .WORD I2
          .BYTE READ,INSLOT
            .
            .      set parameters to process I1, O1
            .
          JSR PC,@(6)+       ;RETURN TO PROCESS AT B+2
          IOT                ;WRITE FROM O1
          .WORD O1
          .BYTE WRITE,OUTSLOT
          IOT                ;READ INTO I1
          .WORD I1
          .BYTE READ,INSLOT
            .
            .      set parameters to process I2, O2
            .
          JSR PC,@(6)+       ;RETURN TO PROCESS AT B+2
          IOT                ;WRITE FROM O2
          .WORD O2
          .BYTE WRITE, OUTSLOT
          BR A               ;READ INTO I2
```

6. The trap handler, below, simulates a two-word JSR instruction with a one-word TRAP instruction. In this example, all TRAP instructions in the program take an operand, and trap to the handler address at location 34. The table of subroutine addresses (e.g., A, B, ...) can be constructed as follows;

```
TABLE:
          CALA=.-TABLE
          .WORD A            ;CALLED BY:  TRAP CALA

          CALB=.-TABLE
          .WORD B            ;CALLED BY:  TRAP CALB
            .
            .
            .
```

Another way to construct the table:

```
TABLE:
          CALA=.-TABLE+TRAP
          .WORD A            ;CALLED BY:  CALA
            .
            .
            .
```

The TRAP handler for either of the above methods follows:

```
TRAP34:  MOV @SP,2(SP)     ;REPLACE STACKED PS WITH PC[1].
         SUB #2,@SP        ;GET POINTER TO TRAP INSTRUCTION.
         MOV @(SP)+,-(SP)  ;REPLACE ADDRESS OF TRAP WITH
                           ;  TRAP INSTRUCTION ITSELF.
         ADD #TABLE-TRAP,@SP ;CALCULATE SUBROUTINE ADDR.
         MOV @(SP)+,PC     ;JUMP TO SUBROUTINE.
```

In the example above, if the third instruction had been written MOV @(SP),(SP) it would have taken an extra word since @(SP) is in Index Mode and assembles as @0(SP). In the final instruction, a jump was executed by a MOV @(SP)+,PC because no equivalent JMP instruction exists.

Following are some JMP and MOV equivalences (note that JMP does not affect condition codes).

JMP (R4)	=	MOV R4,PC
JMP @(R4) (2 words)	=	MOV (R4),PC (1 word)
none	=	MOV @(R4),PC
JMP -(R4)	=	none
JMP @(R4)+	=	MOV (R4),PC
JMP @-(R4)	=	MOV -(R4),PC
none	=	MOV @(R4)+,PC
none	=	MOV @-(R4),PC
JMP X	=	MOV #X,PC
JMP @X	=	MOV X,PC
none	=	MOV @X,PC

The TRAP handler can be useful, also, as a patching technique. Jumping out to a patch area is often difficult because a two-word jump must be performed. However, the one-word TRAP instruction may be used to dispatch to patch areas. A sufficient number of slots for patching should first be reserved in the dispatch table of the TRAP handler. The jump can then be accomplished by placing the address of the patch area into the table and inserting the proper TRAP instruction where the patch is to be made.

[1] Replacing the saved PS loses the T-bit status. If a breakpoint has been set on the TRAP instruction, ODT will not gain control again to reinsert the breakpoints because the T-bit trap will not occur.

APPENDIX A

ASCII CHARACTER SET

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
0	000	NUL	NULL, TAPE FEED, CONTROL SHIFT P.
1	001	SOH	START OF HEADING; ALSO SOM, START OF MESSAGE, CONTROL A,
1	002	STX	START OF TEXT; ALSO EOA, END OF ADDRESS, CONTROL B,
0	003	ETX	END OF TEXT: ALSO EOM, END OF MESSAGE CONTROL C,
1	004	EOT	END OF TRANSMISSION (END): SHUTS OFF TWX MACHINES, CONTROL D,
0	005	ENQ	ENQUIRY (ENQRY); ALSO WRU, CONTROL E,
0	006	ACK	ACKNOWLEDGE. ALSO RU, CONTROL F.
1	007	BEL	RINGS THE BELL. CONTROL G.
1	010	BS	BACKSPACE: ALSO FEO, FORMAT EFFECTOR. BACKSPACE SOME MACHINES, CONTROL H.
0	011	HT	HORIZONTAL TAB. CONTROL I.
0	012	LF	LINE FEED OR LINE SPACE (NEW LINE): ADVANCES PAPER TO NEXT LINE, DUPLICATED BY CONTROL J.
1	013	VT	VERTICAL TAB (VTAB). CONTROL K.
0	014	FF	FORM FEED TO TOP OF NEXT PAGE (PAGE). CONTROL L.
1	015	CR	CARRIAGE RETURN TO BEGINNING OF LINE. DUPLICATED BY CONTROL M.
1	016	SO	SHIFT OUT: CHANGES RIBBON COLOR TO RED. CONTROL N.
0	017	SI	SHIFT IN: CHANGES RIBBON COLOR TO BLACK. CONTROL O
1	020	DLE	DATA LINK ESCAPE. CONTROL P (DC0).
0	021	DC1	DEVICE CONTROL 1, TURNS TRANSMITTER (READER) ON, CONTROL Q (XON).
0	022	DC2	DEVICE CONTROL 2, TURNS PUNCH OR AUXILIARY ON. CONTROL R (TAPE, AUX ON).
1	023	DC3	DEVICE CONTROL e, TURNS TRANSMITTER (READER) OFF, CONTROL S (XOFF).
0	024	DC4	DEVICE CONTROL 4. TURNS PUNCH OR AUXILIARY OFF. CONTROL T (TAPE, AUX OFF)
1	025	NAK	NEGATIVE ACKNOWLEDGE: ALSO ERR. ERROR. CONTROL U.
1	026	SYN	SYNCHRONOUS IDLE (SYNC). CONTROL V.
0	027	ETB	END OF TRANSMISSION BLOCK: ALSO LEM. LOGICAL END OF MEDIUM. CONTROL W.
0	030	CAN	CANCEL (CANCL). CONTROL X.
1	031	EM	END OF MEDIUM. CONTROL Y.
1	032	SUB	SUBSTITUTE. CONTROL Z.
0	033	ESC	ESCAPE. PREFIX.

ASCII CHARACTER SET

1	034	FS	FILE SEPARATOR. CONTROL SHIFT L.
0	035	GS	GROUP SEPARATOR. CONTROL SHIFT M.
0	036	RS	RECORD SEPARATOR. CONTROL SHIFT N.
1	037	US	UNIT SEPARATOR. CONTROL SHIFT O.
1	040	SP	SPACE.
0	041	!	
0	042	"	
1	043	#	
0	044	$	
1	045	%	
1	046	&	
0	047	'	ACUTE ACCENT OR APOSTROPHE.
0	050	(	
1	051	)	
1	052	*	
0	053	+	
1	054	,	
0	055	-	
0	056	.	
1	057	/	
0	060	0	
1	061	1	
1	062	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	067	7	
1	070	8	
0	071	9	
0	072	:	
1	073	;	
0	074	<	
1	075	=	
1	076	>	
0	077	?	
1	100	@	
0	101	A	
0	102	B	
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	
0	110	H	
1	111	I	
1	112	J	
0	113	K	
1	114	L	
0	115	M	
0	116	N	
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	
0	131	Y	
0	132	Z	
1	133	[	SHIFT K

ASCII CHARACTER SET

0	134		SHIFT L
1	135	]	SHIFT M
1	136	↑	SHIFT N
0	137	←	
0	140		ACCENT GRAVE.
0	175		THIS CODE GENERATED BY ALT MODE.
0	176		THIS CODE GENERATED BY ESC KEY (IF PRESENT)
1	177	DEL	DELETE, RUB OUT.
			LOWER CASE ALPHABET FOLLOWS (TELETYPE MODEL 37 ONLY).
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	
1	171	y	
1	172	z	
0	173		
1	174		

APPENDIX B

PAL-11S ASSEMBLY LANGUAGE AND ASSEMBLER

B.1 TERMINATORS

The list below defines all characters which are considered to be terminators. The order of the list implies the descending hierarchy of significance.

Character	Function
CTRL/FORM	Source line terminator.
LINE FEED	Source line terminator.
RETURN	Source line terminator
:	Label terminator
=	Direct assignment delineator
%	Register term delineator
TAB	Item terminator Field terminator
BLANK or SPACE	Item terminator Field terminator
#	Immediate expression field indicator
@	Deferred addressing indicator
(	Initial register field indicator
)	Terminal register field indicator
,	Operand field separator
;	Comments field delimiter
+	Arithmetic addition operator
-	Arithmetic subtraction operator
&	Logical AND operator
!	Logical OR operator
"	Double ASCII text indicator
'	Single ASCII text indicator.

B.2 ADDRESS MODE SYNTAX

r is an integer between 0 and 7.

R is a register expression, E is an expression, ER is either a register expression or an absolute expression in the range of 0 to 7.

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

B.3 INSTRUCTIONS

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

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
DE	Destination effective address
()	contents of
→	becomes
PS	Processor Status word

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 result had an overflow
C	Carry bit:	set if the result had 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.

Logical operators are represented by the following symbols:

!	Inclusive OR
$\textcircled{!}$	Exclusive OR
&	AND
-	(used over a symbol) NOT (i.e., 1's complement)

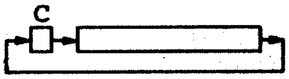
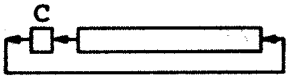
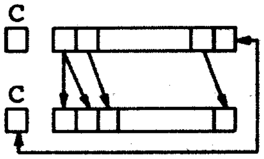
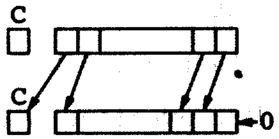
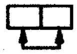
B.3.1 Double Operand Instructions OP A,A

Op-code	MNEMONIC	Stands for	Operation	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	($\overline{SE}$) & (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)→DE	*	*	*	*

B.3.2 Single Operand Instructions OP A

Op-code	MNEMONIC	Stands for	Operation:	Condition Codes N	Z	V	C
0050DD	CLR	CLeaR	0→DE	0	1	0	0
1050DD	CLRB	CLeaR Byte					
0051DD	COM	COMplement	($\overline{DE}$)→DE	*	*	0	1
1051DD	COMB	COMplement Byte					
0052DD	INC	INCrement	(DE) + 1→DE	*	*	*	1
1052DD	INCB	INCrement Byte					
0053DD	DEC	DECrement	(DE) -1→DE	*	*	*	-
1063DD	DECB	DECrement Byte					
0054DD	NEG	NEGate	(DE) + 1→DE	*	*	*	*
1054DD	NEGB	NEGate Byte					
0055DD	ADC	ADd Carry	(DE) + (C)→DE	*	*	*	*
1055DD	ADCB	ADd Carry Byte					
0056DD	SBC	SuBtract Carry	(DE)-(C)→DE	*	*	*	*
1056DD	SBCB	SuBtract Carry Byte					
0057DD	TST	TeST	(DE) - 0→DE	*	*	0	0
1057DD	TSTB	TeST Byte					

B.3.3 Rotate/Shift

0060DD	ROR	ROtate Right		*	*	*	*
1060DD	RORB	ROtate Right Byte		*	*	*	*
0061DD	ROL	ROtate Left		*	*	*	*
1061DD	ROLB	ROtate Left Byte		*	*	*	*
0062DD	ASR	Arithmetic Shift Right		*	*	*	*
1062DD	ASRB	Arithmetic Shift Right Byte		*	*	*	*
0063DD	ASL	Arithmetic Shift Left		*	*	*	*
1063DD	ASLB	Arithmetic Shift Left Byte		*	*	*	*
0001DD	JMP	JuMP	DE→PC	-	-	-	-
0003DD	SWAB	SWAp Bytes		*	*	0	0

B.3.4 Operation Instructions Op

				Condition	Codes		
Op-Code	MNEMONIC	Stands for	Operation	N	Z	V	C
000000	HALT	HALT	The computer stops all functions.	-	-	-	-
000001	WAIT	WAIT	The computer stops and waits for an interrupt.	-	-	-	-
000002	RTI	ReTurn from Interrupt	The PC and PS are popped off the SP stack: ((SP))→PC (SP)+2→SP ((SP))→PS	-	-	-	-
000003	000003	breakpoint trap	Trap to location 14. This is used to call ODT-11.	*	*	*	*
000004	IOT	Input/Output Trap	Trap to location 20. This is used to call IOX.	*	*	*	*
000005	RESET	RESET	Returns all I/O device handlers to power-on state.	-	-	-	-

Trapping	Op or Op	E where $0\leq E\leq 377_8$					
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.	*	*	*	*

CONDITION CODE OPERATES

Op-code	MNEMONIC	Stands for
000241	CLC	CLear Carry bit in PS.
000261	SEC	SEt Carry bit.
000252	CLV	CLear oVerflow bit.
000262	SEV	SEt oVerflow bit.
000244	CLZ	CLear Zero bit.
000264	SEZ	SEt Zero bit.
000250	CLN	CLear Negative bit.
000270	SEN	SEt Negative bit.
000254	CNZ	CLear Negative and Zero bits.
000257	CCC	CLear all Condition Codes.
000277	SCC	Set all Condition Codes.
000240	NOP	No-operation.

B.3.5 Branch Instructions Op E Where $-128_{10}<(E-.-2)/2<127_{10}$

Op-Code	MNEMONIC	Stands for	Condition to be met if branch is to occur
0004XX	BR	BRanch always	
0010XX	BNE	Branch if Not Equal to Zero	$Z=0$
0014XX	BEQ	Branch if EQual (to zero)	$Z=1$
0020XX	BGE	Branch if Greater than or equal (to zero)	$N\textcircled{!}V=0$
0024XX	BLT	Branch if Less Than (zero)	$N\textcircled{!}V = 1$
0030XX	BGT	Branch if Greater Than (zero)	$Z!(N\textcircled{!}V)=0$
0034XX	BLE	Branch if Less than or Equal (to zero)	$Z!(N\textcircled{!}V)=1$

1000XX	BPL	Branch if PLUS	N=0
1004XX	BMI	Branch if MInus	N=1
1010XX	BHI	Branch if HIgher	C!Z=0
1014XX	BLOS	Branch if LOwer or Same	C!Z=1
1020XX	BVC	Branch if oVerflow Clear	V=0
1024XX	BVS	Branch if oVerflow Set	V=1
1030XX	BCC (or BHIS)	Branch if Carry Clear (or Branch if HIgh or Same)	C=0
1034XX	BCS (or BLO)	Branch if Carry Set (or Branch if LOw)	C=1

B.3.6 Subroutine Call JSR ER,A

Op-code	MNEMONIC	Stands for	Operation
004RDD	JSR	Jump to Sub-Routine	Push register on the SP stack, put the PC in the register: DE → TEMP -a temporary storage register internal to processor (SP)-2→SP (REG) → (SP) (PC)+m REG -m depends upon the address mode. (TEMP) → PC

B.3.7 Subroutine Return

Op-code	MNEMONIC	Stands for	Operation
00020R	RTS	ReTurn from Subroutine	Put register contents in PC and pop old contents from SP stack into register.

B.3.8 Extensions for the LSI-11 Version Of PAL-11S

Op-code	MNEMONIC	Stands for	Operation	Condition Codes			
				N	Z	V	C
0067dd	SXT	Sign eXTend	Nx(-1) DE	-	*	0	-
1067dd	MFPS	Move byte From PS	(PS) DE	*	*	0	-

[1] These extensions are available only with the LSI-11 version of PAL-11S.

1064ss	MTPS	Move byte To PS	(SE) PS	*	*	*	*
074rdd	XOR	eXclusive OR	r ! (DE) DE	*	*	0	-
070rss	MUL	MULtiply	r x (SE) r	*	*	0	*
071rss	DIV	DIVide	r / (SE) r	*	*	*	*
072rss	ASH	Arithmetic SHift		*	*	*	*
073rss	ASHC	Arithmetic SHift Combined		*	*	*	*
0064nn	MARK	MARK	SP+2xnn SP R5 PC SP^ R5	-	-	-	-
077rnn	SOB	Subtract One and Branch if 0	(r)-1 r; if (r) 0 then (PC)-2xnn PC	-	-	-	-
000006	RTT	ReTurn from Trap	((SP)) PC ((SP))+2 SP ((SP)) PS (SP)+2 SP	loaded from stack			

B.4 ASSEMBLER DIRECTIVES

MNEMONIC	Operand	Stands for	Operation
.EOT	none	End Of Tape	Indicates the physical end of the source input medium
.EVEN	none	EVEN	Insures that the assembly location counter is even by adding 1 if it is odd.
.END	E (E optional)	END	Indicates the physical and logical end of the program and optionally specifies the entry point (E)
.WORD	E,E,...	WORD	Generates words of data
	E,E,...	(the void operator)	Generates words of data
.BYTE	E,E...	BYTE	Generates bytes of data
.ASCII	/xxx...x/	ASCII	Generates 7-bit ASCII characters for text enclosed by delimiters.
.TITLE	NAME	TITLE	Generates a name for the object module.
.ASECT	none	ASECT	Initiates the Absolute section.
.CSECT	none	CSECT	Initiates the Relocatable Control section.
.LIMIT	none	LIMIT	Generates two words containing the

			low and high limits of the relocatable section.
.GLOBL	NAME,NAME,...	GLOBAL	Specifies each name to be a global symbol
.RAD50	/XXX/	RADIX 50	Generates the RADIX 50 representation of the ASCII character in delimiters.
.IFZ	E	IF E=0	Assemble what follows up to the terminating .ENDC if the expression E is 0.
.IFNZ	E	IF E≠0	Assemble what follows up to the terminating .ENDC, if the expression E is not 0.
.IFL	E	IF E<0	Assemble what follows up to the terminating .ENDC, if the expression E is less than 0.
.IFLE	E	IF E≤0	Assemble what follows up to the terminating .ENDC, if the expression E is less than or equal to 0.
.IFG	E	IF E>0	Assemble what follows up to the terminating .ENDC, if the expression E is greater than 0.
.IFGE	E	IF E≥0	Assemble what follows up to the terminating .ENDC, if the expression E is greater than or equal to 0.
.IFDF	NAME	IF NAME defined	Assemble what follows up to the terminating .ENDC if the symbol NAME is defined.
.IFNDF	NAME	IF NAME undefined	Assemble what follows up to the terminating .ENDC if the symbol NAME is undefined.
.ENDC	none	End of Conditional	Terminates the range of a conditional directive.

B.5 ERROR CODES

Error Code	Meaning
A	Addressing error. An address within the instruction is incorrect. Also includes relocation errors.
B	Bounding error. Instructions or word data are being assembled at an odd address in memory.
D	Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.
I	Illegal character detected. Ilegal characters which are also non-printing are replaced by a ? on the listing.
L	Line buffer overflow. All extra characters beyond 72 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 an 8 or 9 was not terminated by a decimal point.
P	Phase error. A label's definition or 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 followed by a linefeed or form feed.
R	Register-type error. An invalid use of or reference to a register has been made.
S	Symbol table overflow. When the quantity of user-defined symbols exceeds the allocated space available in the user's symbol table, the assembler outputs the current source line with the S error code, then returns to the command string interpreter to await the next command string to be typed.
T	Truncation error. More than the allotted number of bits were input so the leftmost bits were truncated. T error does not occur for the result of an expression.
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.

B.6 INITIAL OPERATING PROCEDURES

Loading: Use Absolute Loader. The start address of the Loader must be in the console switches.

Storage Requirements:PAL-11S uses 8K of memory.

Starting: Immediately upon loading, PAL-11S will be in control and initiate dialogue.

Initial Dialogue:

Printout	Inquiry
*S	What is the input device of the Source symbolic tape?
*B	What is the output device of the Binary object tape?
*L	What is the output device of the assembly Listing?
*T	What is the output device of the symbol Table?

Each of these questions may be answered by any one of the following characters:

Character	Answer Indicated
T	Teleprinter keyboard
L	Low-speed reader or punch
H	High-speed reader or punch
P	Line Printer

Each of these answers may be followed by the other characters indicating options:

Option Typed	Function to be performed
/1	on pass 1
/2	on pass 2
/3	on pass 3
/E	errors to be listed on the Teletype on the same pass (meaningful only for *B or *L).

Each answer is terminated by typing the RETURN key. A RETURN alone as answer will delete the function.

Dialogue During Assembly:

Printout	Response
EOF ?	Place next tape in reader and type RETURN. A .END statement may be forced by typing E followed by RETURN.
END ?	Start next pass by placing first tape in reader and typing RETURN.

EOM ? If the end-of-medium is on the listing device, the device may be readied and the assembly may be continued by typing RETURN.

If the end-of-medium is on the binary device, the assembler will discontinue the assembly and restart itself.

Restarting: Type CTRL/P. The initial dialogue will be started again.

APPENDIX C

PAL-11A ASSEMBLY LANGUAGE AND ASSEMBLER

C.1 SPECIAL CHARACTERS

Character	Function
form feed	Source line terminator
line feed	Source line terminator
carriage return	Source statement 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
,	Operand field separator
;	Comment field indicator
+	Arithmetic addition operator
-	Arithmetic subtraction operator
&	Logical AND operator
!	Logical OR operator
"	Double ASCII character indicator
'	Single ASCII character indicator
.	Assembly location counter

C.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.

C.3 INSTRUCTIONS

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

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
DE	Destination Effective address
()	Contents of
→	Is transferred to
PS	Processor Status word

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
$\textcircled{!}$	Exclusive OR
&	AND
-	(used over a symbol) NOT (i.e., 1's complement)

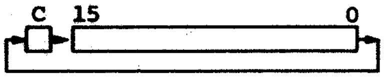
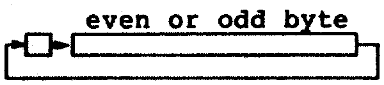
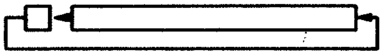
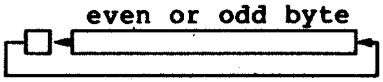
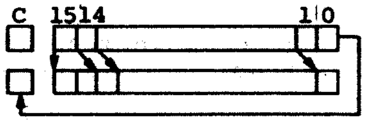
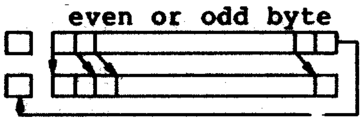
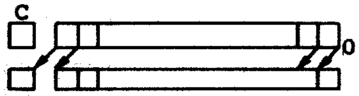
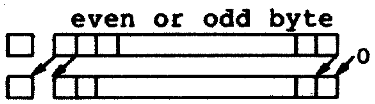
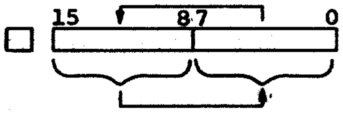
C.3.1 Double-Operand Instructions Op A,A

				Status Word Condition Codes			
Op-Code	MNEMONIC	Stands for	Operation	N	Z	V	C
01SSDD	MOV	MOVe	(SE) → DE	*	*	0	-
11SSDD	MOVB	MOVe Byte					
02SSDD	CMP	CoMPare	(SE)-(DE)	*	*	*	*
12SSDD	CMPB	CoMPare Byte					
03SSDD	BIT	BIt Test	(SE)&(DE)	*	*	0	-
13SSDD	BITB	BIt Test Byte					
04SSDD	BIC	BIt Clear	$(\overline{SE})$&(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) → DE	*	*	*	*

C.3.2 Single-Operand Instructions Op A

				Status Word Condition Codes			
Op-Codes	MNEMONIC	Stands for	Operation	N	Z	V	C
0050DD	CLR	CLeaR	0 DE	0	1	0	0
1050DD	CLRB	CLeaR Byte					
0051DD	COM	COMplement	$(\overline{DE})$ DE	*	*	0	1
1051DD	COMB	COMplement Byte					
0052DD	INC	INCrement	(DE)+1 DE	*	*	*	-
1052DD	INCB	INCrement Byte					
0053DD	DEC	DECrement	(DE)-1 DE	*	*	*	-
1053DD	DECB	DECrement Byte					
0054DD	NEG	NEGate	$(\overline{DE})$+1 DE	*	*	*	*
1054DD	NEGB	NEGate Byte					
0055DD	ADC	ADd Carry	(DE)+(C) → DE	*	*	*	*
1055DD	ADCB	ADd Carry Byte					
0056DD	SBC	SuBtract Carry	(DE)-(C) → DE	*	*	*	*
1056DD	SBCB	SuBtract Carry Byte					
0057DD	TST	TeST	(DE)-0 → DE	*	*	0	0
1057DD	TSTB	TeST Byte					

C.3.3 Rotate/Shift Instructions Op A

				Status Word Condition Codes			
Op-Code	MNEMONIC	Stands for	Operation	N	Z	V	C
0060DD	ROR	ROtate Right		*	*	*	*
1060DD	RORB	ROtate Right Byte		*	*	*	*
0061DD	ROL	ROtate Left		*	*	*	*
1061DD	ROLB	ROtate Left Byte		*	*	*	*
0062DD	ASR	Arithmetic Shift Right		*	*	*	*
1062DD	ASRB	Arithmetic Shift Right Byte		*	*	*	*
0063DD	ASL	Arithmetic Shift Left		*	*	*	*
1063DD	ASLB	Arithmetic Shift Left Byte		*	*	*	*
0001DD	JMP	JuMP	DE → PC	-	-	-	-
0003DD	SWAB	SWAp Bytes		*	*	0	0

C.3.4 Operate Instructions Op

				Status Word Condition Codes			
Op-Code	MNEMONIC	Stands for	Operation	N	Z	V	C
000000	HALT	HALT	The computer stops all functions.	-	-	-	-
000001	WAIT	WAIT	The computer stops and waits for an interrupt.	-	-	-	-

000002	RTI	ReTurn from Interrupt	The PC and PS are popped off the SP stack: ((SP))→PC (SP)+2→SP ((SP))→PS (SP)+2→SP RTI is also used to return from a trap.	*	*	*	*
000005	RESET	RESET	Returns all I/O devices to power-on state.	-	-	-	-

C.3.5 Trap Instructions Op or Op E Where 0≤E≤377_8

				Status Word			
				Condition Codes			
Op-Code	MNEMONIC	Stands for	Operation	N	Z	V	C
*000003	(none)	(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.	*	*	*	*

*Op (only)

CONDITION CODE OPERATES

Op-Code	MNEMONIC	Stands for
000241	CLC	CLear Carry bit in PS.
000261	SEC	SEt Carry bit.
000242	CLV	CLear oVerflow bit.
000262	SEV	SEt oVerflow bit.
000244	CLZ	CLear Zero bit.
000264	SEZ	SEt Zero bit.
000250	CLN	CLear Negative bit.
000270	SEN	SEt Negative bit.
000254	CNZ	CLear Negative and Zero bits.
000257	CCC	Clear all Condition Codes.
000277	SCC	Set all Condition Codes.

C.3.6 Branch Instructions 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 EQual (to zero)	Z=1
0020XX	BGE	Branch if Greater than or Equal (to zero)	$N \textcircled{!} V=0$
0024XX	BLT	Branch if Less Than (zero)	$N \textcircled{!} V=1$
0030XX	BGT	Branch if Greater Than (zero)	$Z ! (N \textcircled{!} V)=0$
0034XX	BLE	Branch if Less than or Equal (to zero)	$Z ! (N \textcircled{!} V)=1$
1000XX	BPL	Branch if PLus	N=0
1004XX	BMI	Branch if MInus	N=1
1010XX	BHI	Branch if HIgher	C ! Z = 0
1014XX	BLOS	Branch if LOwer or Same	C ! Z = 1
1020XX	BVC	Branch if oVerflow Clear	V=0
1024XX	BVS	Branch if oVerflow Set	V=1
1030XX	BCC (or BHIS)	Branch if Carry Clear (or Branch if HIgher or Same)	C=0
1034XX	BCS (or BLO)	Branch if Carry Set (or Branch if LOwer)	C=1

C.3.7 Subroutine Call Op ER, A

Op-Code	MNEMONIC	Stands for	Operation
004RDD	JSR	Jump to SubRoutine	Push register on the SP stack,put the PC in the register: DE (TEMP) - a temporary storage register internal to processor. (SP)-2 → SP (REG) → (SP) (PC) → REG (TEMP) → PC

C.3.8 Subroutine Return Op ER

Op-Code	MNEMONIC	Stands for	Operation
00020R	RTS	ReTurn from Subroutine	Put register contents into PC and pop old contents from SP stack into register.

C.4 ASSEMBLER DIRECTIVES

Op-Code	MNEMONIC	Stands for	Operation
	.EOT	End Of Tape	Indicates the physical end of the source input medium
	.EVEN	EVEN	Ensures that the assembly location counter is even by adding 1 if it is odd
	.END m (m optional)	END	Indicates the physical and logical end of the program and optionally specifies the entry point (m)
	.WORD E,E,..	WORD .	Generates words of data
	E,E,...	(the void operator)	Generates words of data
	.BYTE E,E,...	BYTE	Generates bytes of data
	.ASCII /xxx...x/	ASCII	Generates 7-bit Ascii character for the text enclosed by delimiters

C.5 ERROR CODES

Error Code — Meaning

A — Addressing error. An address within the instruction is incorrect.

B — Bounding error. Instructions or word data are being assembled at an odd address in memory.

D — Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.

I — Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing.

L — Line buffer overflow. Extra characters (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 an 8 or 9 has a decimal point missing.

P — Phase error. A label's definition or 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 followed by a line feed or form feed.

R Register-type error. An invalid use of or reference to a register has been made.

S Symbol-table overflow. When the quantity of user-defined symbols exceeds the allocated space available in the user's symbol table, the assembler outputs the current source line with the S error code, then returns to the command string interpreter to await the next command string to be typed.

T Truncation error. A number was too big for the allotted number of bits; the leftmost bits were truncated. T error does not occur for the result of an expression.

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.

C.6 INITIAL OPERATING PROCEDURES

Loading: Use Absolute Loader (see Chapter 6). Make sure that the start address of the absolute loader is in the switches when the assembler is loaded.

Storage Requirements: PAL-11A exists in 4K and 8K versions.

Starting Immediately upon loading, PAL-11A will be in control and initiate dialogue.

Initial Dialogue:

Printout	Inquiry
*S	What is the input device of the Source symbolic tape?
*B	What is the output device of the Binary object tape?
*L	What is the output device of the assembly Listing?
*T	What is the output device of the symbol Table?

Each of these questions may be answered by one of the following characters:

Character	Answer Indicated
T	Teletype keyboard
L	Low-speed reader or punch
H	High-speed reader or punch
P	line Printer (8K version only)

Each of these answers may be followed by other characters indicating options:

Option Typed	Function to be Performed
/1	on pass 1
/2	on pass 2
/3	on pass 3
/E	errors to be listed on the Teletype on the same pass (meaningful or *B or *L only)

Each answer is terminated by typing the RETURN key. A RETURN alone as answer will delete the function.

Dialogue during assembly:

Printout	Response
EOF ?	Place next tape in reader and type RETURN. A .END statement may be forced by typing E followed by RETURN.
END ?	Start next pass by placing first tape in reader and typing RETURN.
EOM ?	If listing on HSP or LPT, replenish tape or paper and type RETURN. If binary on HSP, start assembly again.
Restarting:	Type CTRL/P. The initial dialogue will be started again.

APPENDIX D

TEXT EDITOR, ED-11

D.1 INPUT/OUTPUT COMMANDS

R — Reads a page of text from input device, and appends it to the contents (if any) of the page buffer. Dot is moved to the beginning of the page and Marked. (See B and M below.)

O — Opens the input device when the user wishes to continue input with a new tape in the reader.

n, -n, 0, @, / } L — Lists the character string

n, -n, 0, @, / } P — Punches the character string

ARGUMENTS

(n) beginning at Dot and ending with nth line feed character.

(-n) beginning with 1st character following the (n+1)th previous line feed and terminating at Dot.

(0) beginning with 1st character of current line and ending at Dot.

(@) bounded by Dot and the Marked location (see M).

(/) beginning at Dot and ending with the last character in the page.

F — Outputs a Form Feed character and four inches of blank tape.

nT — Punches four inches of Trailer (blank tape) n times.

nN — Punches contents of the page buffer (followed by a trailer if a form feed is present), deletes the contents of the buffer, and reads the next page into the page buffer. It does this n times. At completion, Dot and Mark are located at the beginning of the page buffer.

V — Lists the entire line containing Dot (i.e., from previous line feed to next line feed or form feed).

< — Same as -1L. If Dot is located at the beginning of a line, this simply lists the line preceding the current line.

> — Lists the line following the current line.

D.2 POINTER-POSITIONING COMMANDS

B Moves Dot to the beginning of the page.

E Moves Dot to the end of the page.

M Marks the current position of Dot for later reference in a command using the argument @. Certain commands implicitly move Mark.

n -n 0 @ /	J	Moves Dot:	(n) (-n) (0) (@) (/)	forward past n characters backward past n characters to the beginning of the current line to the Marked location to the end of the page
n -n 0 @ /	A	Moves Dot:	(n) (-n) (0) (@) (/)	forward past n ends-of-lines to first character following the (n+1)th previous end-of-line to the beginning of current line to the Marked location to the end of the page

D.3 SEARCH COMMANDS

nG
XXXX
Gets (searches for) the nth occurrence of the specified character string on the current page. Dot is set immediately after the last character in the found text, and the characters from the beginning of the line to Dot are listed on the teleprinter. If the search is unsuccessful, Dot will be at the end of the buffer and a ? will be printed out.

H
XXXX
Searches the wHole file for the next occurrence of the specified character string. Combines G and N commands. If search is not successful on current page, it continues on Next page. Dot is set immediately after the last character in the found text and the characters from the beginning of the line to Dot are listed on the teleprinter. If the Search object is not found, Dot will be at the end of the buffer and a ? will be printed out. In such a case, all text scanned is copied to the output tape.

D.4 COMMANDS TO MODIFY THE TEXT

	Character-Oriented			Line-Oriented	
nD nC XXXX	Deletes Changes	the following n characters	nK nX XXXX	Kills eXchanges	the character string beginning at Dot and ending at the nth end-of-line.
-nD -nC XXXX	Deletes Changes	the previous n characters	-nK -nX XXXX	Kills eXchanges	the character string beginning with the first character following the (n+1)th previous end-of-line and ending at Dot.

0D 0C XXXX	Deletes Changes	the current line up to Dot	0K 0X XXXX	Kills eXchanges	the current line up to Dot.
@D @C XXXX	Deletes Changes	The character string beginning at Dot and ending at a previously Marked location.	@K @X XXXX	Kills eXchanges	the character string beginning at Dot and ending at a previously Marked location.
/D /C XXXX	Deletes Changes	the character string beginning at Dot and ending with the last character of the page.	/K /X XXXX	Kills eXchanges	the character string beginning at Dot and ending with the last character of the page.

I
XXXX   Inserts the specified text. LINE FEED terminates Text Mode and causes execution of the command. Dot is set to the location immediately following the last character inserted. If text was inserted before the position of Mark, ED-11 performs an M command.

D.5 SYMBOLS

Dot   Location following the most recent character operated upon.

↑   Holding down the CTRL key (not the ↑ key) in combination with another keyboard character.

RETURN   If in command mode, it executes the current command; goes into Text Mode if required. If in Text Mode, it terminates the current line, enters a carriage return and line feed into the buffer and stays in text mode. At all times causes the carriage to move to the beginning of a new line. (RETURN is often symbolized as ↵).

↓   (Typing the LINE FEED key) Terminates Text Mode unless the first character typed in Text Mode; executes the current command.

CTRL/FORM   A Form feed which terminates, and thus defines, a page of the user's text.

D.6 GROUPING OF COMMANDS

No Arguments		Argument n only		All Arguments (n,-n,0,@,/)	
V	(Verify:	G	(Get)	A	(Advance)
	Lists current line)	N	(Next)	C	(Change)
<	(Lists previous line)	T	(Trailer)	D	(Delete)
>	(Lists next line)			J	(Jump)
B	(Begin)			K	(Kill)
E	(End)			L	(List)
F	(Form feed)			P	(Punch)
H	(wHole)			X	(eXchange)
I	(Insert)				
M	(Mark)				
O	(Open)				
R	(Read)				

Requiring Text Mode	Line Oriented	Character Oriented
C (Change)	A (Advance)	J (Jump)
G (Get)	K (Kill)	D (Delete)
H (wHole)	L (List)	
I (Insert)	P (Punch)	
X eXchange)	X eXchange)	C (Change)

D.7 OPERATING PROCEDURES

D.7.1 Loading: Use Absolute Binary Loader (see Chapter 5).

D.7.2 Storage Requirements: ED-11 uses all of core.

D.7.3 Starting: Immediately upon loading, ED-11 will be in control.

D.7.4 Initial Dialogue:

Program Types		User Response
*I	L↵	(if LSR is to be used for source input)
	H↵	(if HSR is to be used for source input)
*O	L↵	(if LSP is to be used for edited output)
	H↵	(if HSP is to be used for edited output)

If the output device is the high-speed punch (HSP), Editor enters command mode to accept input. Otherwise the sequence continues with:

LSP OFF? ↵(when LSP is off)

Upon input of ↵ from the keyboard, Editor enters command mode and is ready to accept input.

D.7.5 Restarting:

Type CTRL/P twice, initiating the normal initial dialogue. The text to be edited should be loaded (or reloaded) at this time.

APPENDIX E

DEBUGGING OBJECT PROGRAMS ON-LINE, ODT-11 AND ODT-11X

E.1 SUMMARY OF CONTENTS

ODT indicates readiness to accept commands by typing * or by opening a location by printing its contents.

1. ODT-11

n/	opens word n
\	reopens last word opened
RETURN key	closes open location
↓	opens next location
↑	opens previous location
←	opens relatively addressed word
$n/	opens general register n (0-7)
n;G	goes to word n and starts execution
n;B	sets breakpoint at word n
;B	removes breakpoint
$B/	opens breakpoint status word
;P	proceeds from breakpoint, stops again on next encounter
n;P	proceeds from breakpoint, stops again on nth encounter
$M/	opens mask for word search
n;W	searches for words which match n in bits specified in $M
n;E	searches for words which address word n
n/ (contents) m;O	calculates offsets from n to m
$S/	opens location containing user program's status register
$P/	opens location containing ODT's priority level

NOTE

> If a word is currently open, new contents for the word may be typed followed by any of the commands RETURN,↓,↑,← or . The open word will be modified and closed before the new command is executed.

2. ODT-11X

In addition to the commands of the regular version, the extended version has the following:

n\	opens byte
\	reopens last byte opened
@	opens the absolutely addressed word
>	opens the word to which the branch refers
<	opens next location of previous sequence
n;rB	(r between 0 and 7) sets breakpoint r at word n
;rB	removes breakpoint r
;B	removes all breakpoints
$B/	opens breakpoint 0 status word. Successive LINE FEEDs open words for other breakpoints and single-instruction mode.
;nS	enables Single-instruction mode (n can have any value and is not significant)
n;P	in single-instruction mode, Proceeds with program run for next n instructions before reentering ODT (if n is missing, it is assumed to be 1)
;S	disables Single-instruction mode

APPENDIX F

LOADING AND DUMPING CORE MEMORY

F.1 THE BOOTSTRAP LOADER

This appendix pertains only to systems with a Switch Register.

F.1.1 Loading The Bootstrap Loader

The Bootstrap Loader should be toggled into the highest core memory bank.

```
xx7744          016701
xx7746          000026
xx7750          012702
xx7752          000352
xx7754          005211
xx7756          105711
xx7760          100376
xx7762          116162
xx7764          000002
xx7766          xx7400
xx7770          005267
xx7772          177756
xx7774          000765
xx7776          yyyyyy
```

xx represents the highest available memory bank. For example, the first location of the loader would be one of the following, depending on memory size, and xx in all subsequent locations would be the same as the first.

Location	Memory Bank	Memory Size
017744	0	4K
037744	1	8K
057744	2	12K
077744	3	16K
117744	4	20K
137744	5	24K
157744	6	28K

The contents of location xx7776 (yyyyyy) in the Instruction column above should contain the device status register address of the paper tape reader to be used when loading the bootstrap formatted tapes specified as follows:

Teletype Paper Tape Reader -- 177560

High-speed Paper Tape Reader -- 177550

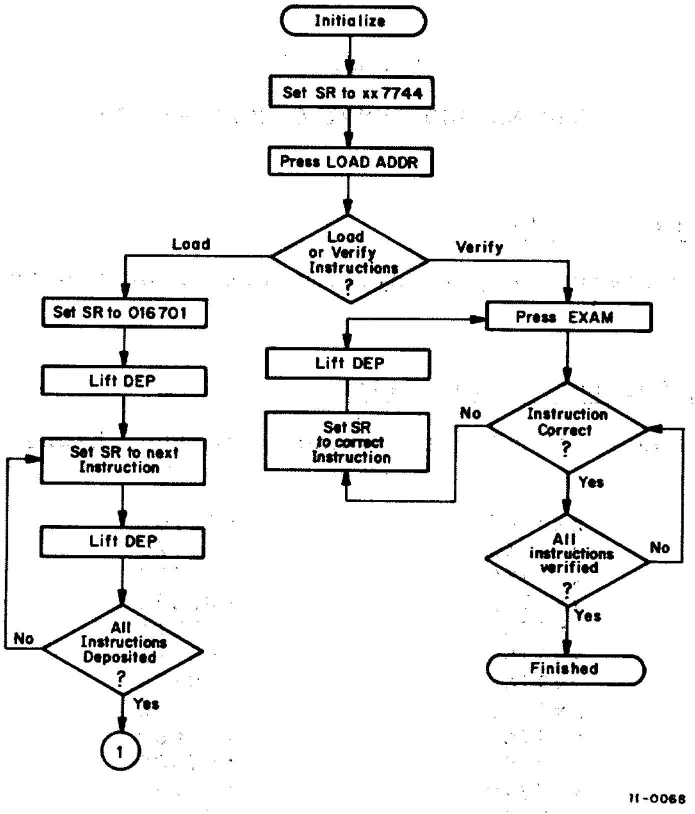


Figure F-1 Loading and Verifying the Bootstrap Loader

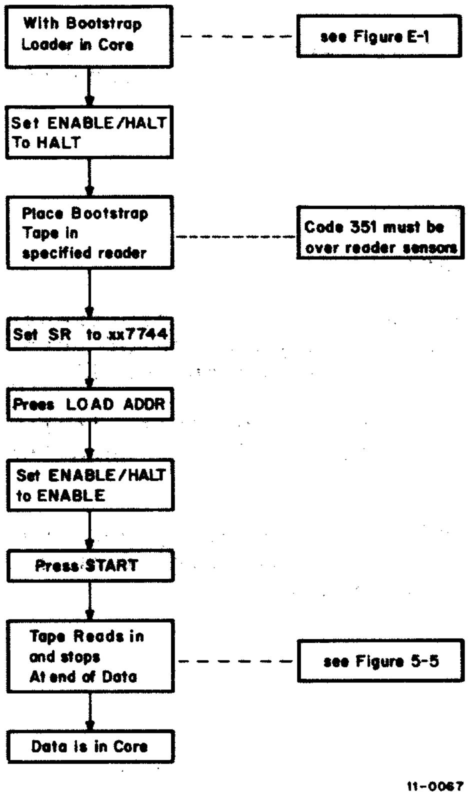

With Bootstrap Loader in Core
see Figure E-1
Set ENABLE/HALT To HALT
Place Bootstrap Tape in specified reader
Code 351 must be over reader sensors
Set SR to xx7744
Press LOAD ADDR
Set ENABLE/HALT to ENABLE
Press START
Tape Reads in and stops At end of Data
see Figure 5-5
Data is in Core
11-0067

Figure F-2 Loading Bootstrap Tapes into Core

F.2 THE ABSOLUTE LOADER

1. Loading the Absolute Loader

   The Bootstrap Loader is used to load the Absolute Loader into core. (See Figure F-2.) The Absolute Loader occupies locations xx7474 through xx7743, and its starting address is xx7500.

2. Loading with the Absolute Loader

   When using the Absolute Loader, there are three types of loads available: normal, relocated to specific address, and continued relocation.

Optional switch register settings for the three types of loads are listed below.

	Switch Register	
Type of load	Bits 1-14	Bit 0
Normal	(ignored)	0
Relocated - continue loading where left off	0	1
Relocated - load in specified area of core	nnnnn (specified address)	1

F.3 CORE MEMORY DUMPS

The two dump programs are

> DUMPTT, which dumps the octal representation of the contents of all or specified portions of core onto the teleprinter, low-speed or high-speed punch, or line printer.
>
> DUMPAB, which dumps the absolute binary code of the contents of specified portions of core onto the low-speed (Teletype) or high-speed punch.

Both dumps are supplied on punched paper tape in bootstrap and absolute binary formats. The following figure summarizes loading and using the Absolute binary tapes.

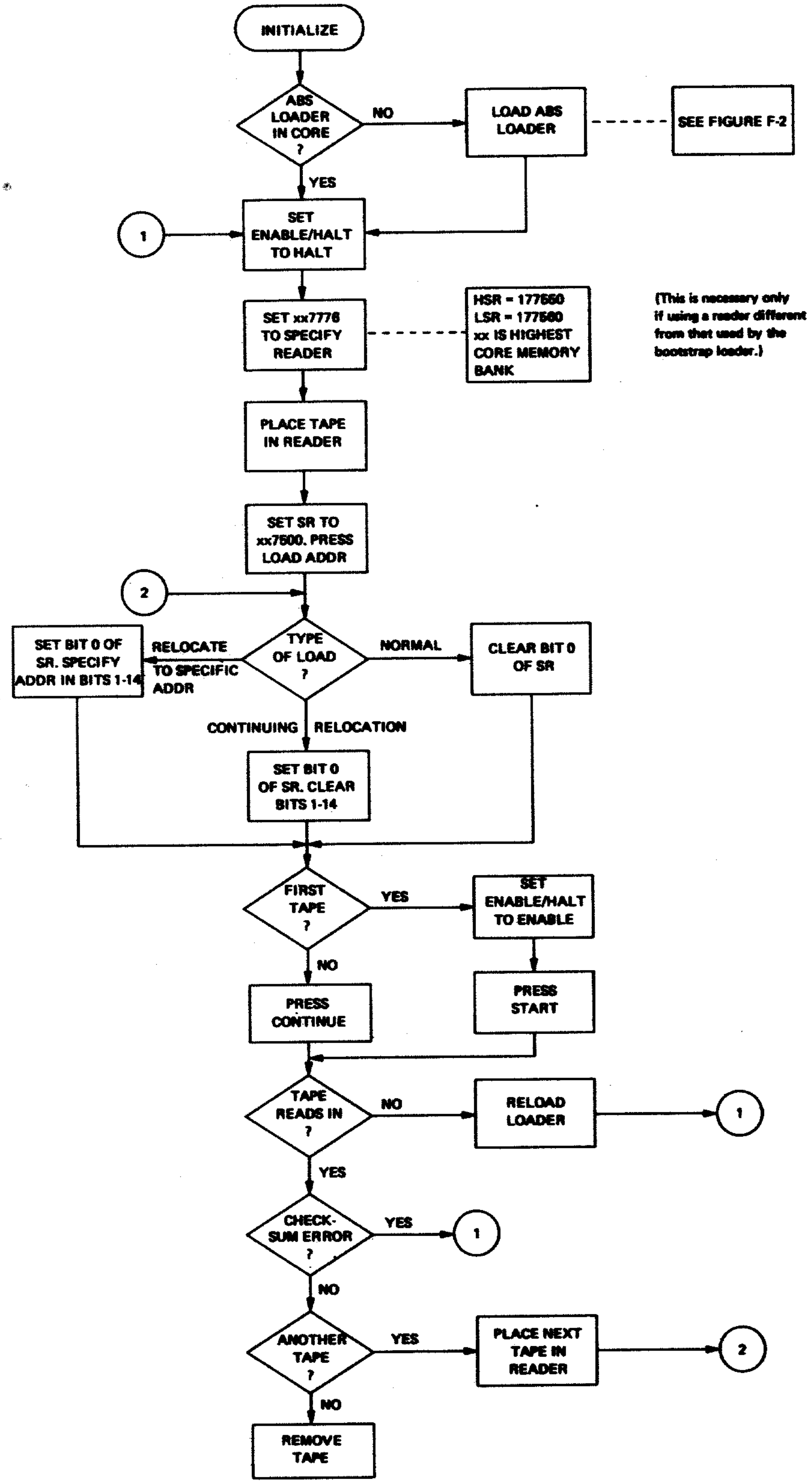

INITIALIZE
ABS LOADER IN CORE ?
NO
LOAD ABS LOADER
SEE FIGURE F-2
YES
1
SET ENABLE/HALT TO HALT
SET xx7776 TO SPECIFY READER
HSR = 177550
LSR = 177560
xx IS HIGHEST CORE MEMORY BANK
(This is necessary only if using a reader different from that used by the bootstrap loader.)
PLACE TAPE IN READER
SET SR TO xx7500. PRESS LOAD ADDR
2
TYPE OF LOAD ?
RELOCATE TO SPECIFIC ADDR
SET BIT 0 OF SR. SPECIFY ADDR IN BITS 1-14
NORMAL
CLEAR BIT 0 OF SR
CONTINUING RELOCATION
SET BIT 0 OF SR. CLEAR BITS 1-14
FIRST TAPE ?
YES
SET ENABLE/HALT TO ENABLE
PRESS START
NO
PRESS CONTINUE
TAPE READS IN ?
NO
RELOAD LOADER
1
YES
CHECK-SUM ERROR ?
YES
1
NO
ANOTHER TAPE ?
YES
PLACE NEXT TAPE IN READER
2
NO
REMOVE TAPE

Figure F-3 Loading with the Absolute Loader

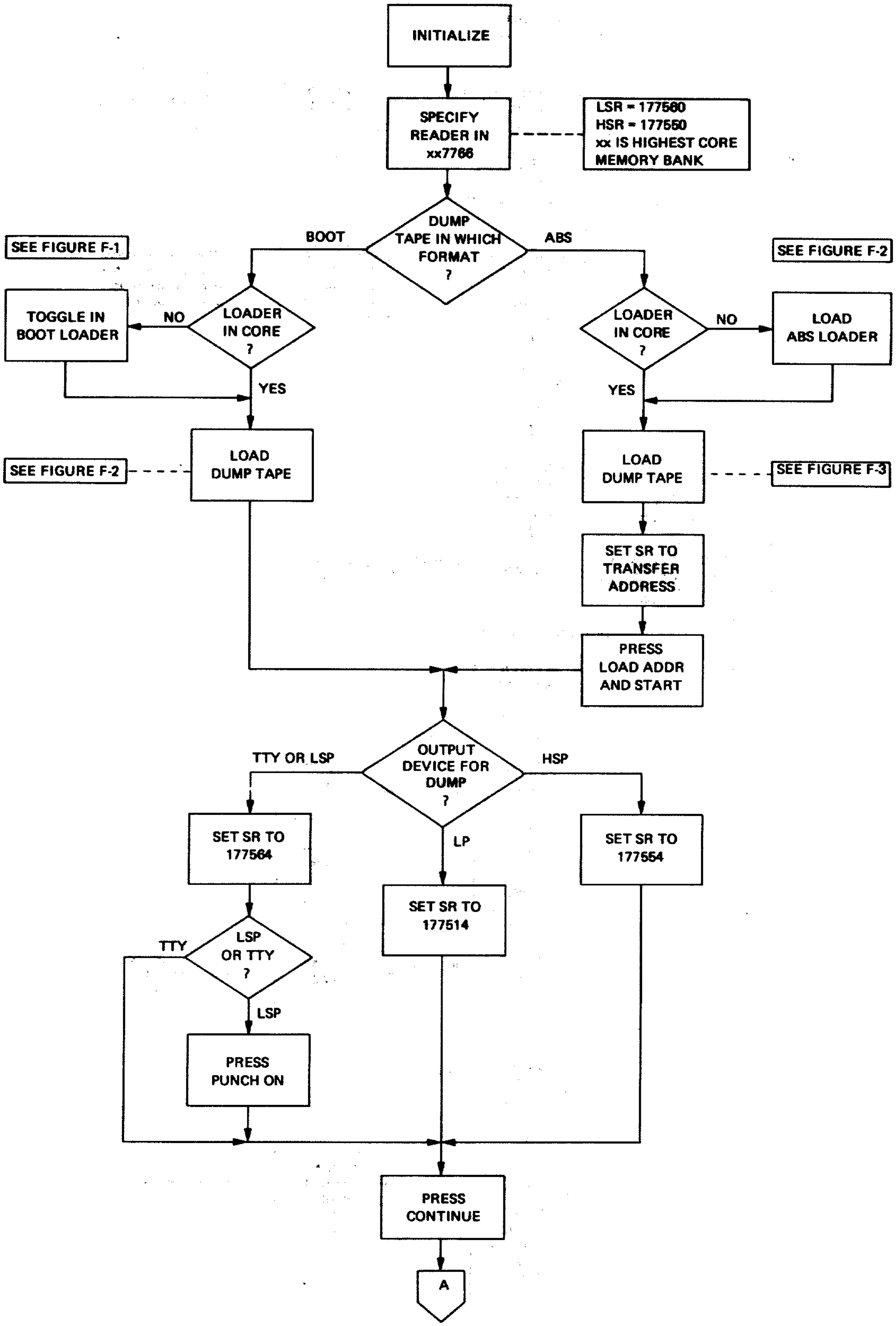


Figure F-4 Dumping Using DUMPAB or DUMPTT

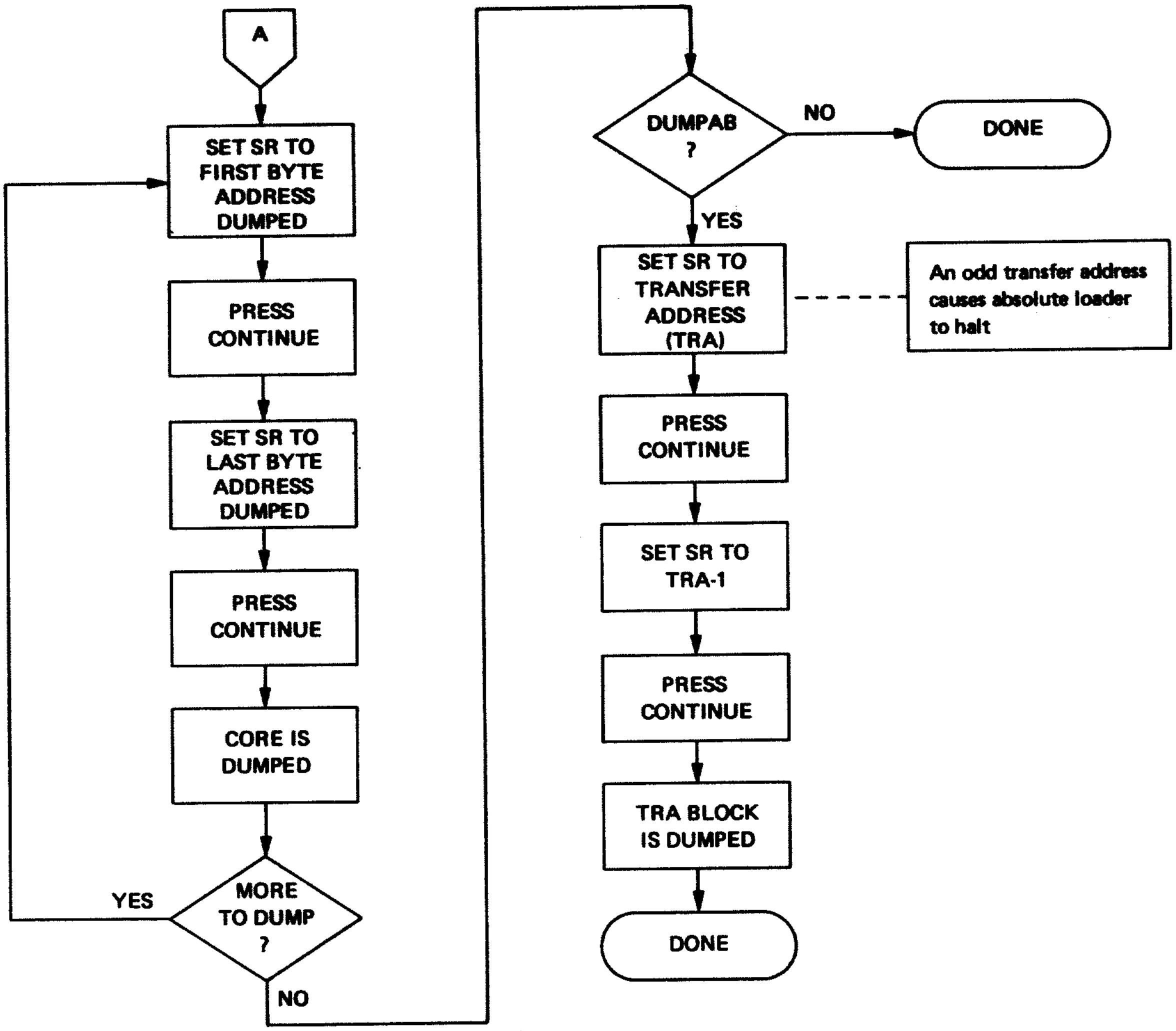

A
SET SR TO FIRST BYTE ADDRESS DUMPED
PRESS CONTINUE
SET SR TO LAST BYTE ADDRESS DUMPED
PRESS CONTINUE
CORE IS DUMPED
MORE TO DUMP ?
YES
NO
DUMPAB ?
NO
DONE
YES
SET SR TO TRANSFER ADDRESS (TRA)
An odd transfer address causes absolute loader to halt
PRESS CONTINUE
SET SR TO TRA-1
PRESS CONTINUE
TRA BLOCK IS DUMPED
DONE

Figure F-4 (continued). Dumping Using DUMPAB or DUMPTT

APPENDIX G

INPUT/OUTPUT PROGRAMMING, IOX

G.1 INSTRUCTION SUMMARY

1. Format

```
IOT
.WORD  (an address)
.BYTE  (a command code, a slot number of a device)
.WORD  (done address)              ;READR AND WRITR ONLY
```

2. Command Codes:

```
 INIT   = 1
RESET   = 2
RSTRT   = 3
WAITR   = 4
 SEEK   = 5
 READ   = 11
WRITE   = 12
READR   = 13
WRITR   = 14
```

G.2 PROGRAM FLOW SUMMARY

1. Set up buffer header:

	Location	Contents
BUFFER HEADER	Buffer and Buffer+1	Maximum number of data bytes (unsigned integer)
	Buffer+2	Mode of data (byte)
	Buffer+3	Status of data (byte)
	Buffer+4 and Buffer+5	Number of data bytes involved in transfer (unsigned integer)
	Buffer+6	Actual data begins here.

Mode Byte Format

Mode Byte Format

Bits	7	6	5	4	3	2	1	0	Bits
1=	NO ECHO						UNFORMATTED	BINARY	=1
0=	ECHO						FORMATTED	ASCII	=0

Coding Mode Byte

Formatted ASCII	0	(or 200 to suppress echo)
Formatted Binary	1	
Unformatted ASCII	2	(or 202 to suppress echo)
Unformatted Binary	3	

Status Byte Format

Status Byte Format

7	6	5	4	3	2	1	0
1 = DONE	1 = EOM	1 = EOF	SEE CODES				
			NON-FATAL ERRORS				

Coding Non-Fatal Errors

2	=	checksum error (formatted binary)
3	=	truncation of a long line
4	=	an improper mode

2. Assign devices to slots in Device Assignment Table:

   (RESET and INIT commands)

   Slot numbers are in the range 0 to 7.

   Device Codes:

KBD = 1	LSP = 4	LPT = 10
TTY = 2	HSR = 5	
LSR = 3	HSP = 6	

3. Use a data transfer command to initiate the transfer.

G.3 FATAL ERRORS

Fatal errors result in a jump to 40 with R0 set to the error code. R1 is set to the value of the PC for error code 0. Errors 1-5 cause R1 to be set to an IOT argument or to the instruction following the arguments.

Fatal Error Code	Reason
0	Illegal Memory Reference, SP overflow, illegal instruction
1	Illegal command
2	Slot out of range
3	Device out of range
4	Slot not inited
5	Illegal data mode

APPENDIX H

SUMMARY OF FLOATING POINT MATH PACKAGE, FPMP-11

This appendix lists all the global entry points of FPMP-11 and provides a brief description of the purposes of each. Sections H.1 and H.2 are for reference when it is desired to call FPMP-11 routines directly (i.e., without the use of the TRAP handler). Entry names preceded by an octal number can be referenced via the TRAP handler. The number is the "routine number" referred to in the FPMP-11 manual. If the number is enclosed in parentheses, the routine cannot be accessed by the present TRAP handler, but has been assigned a number for future use. For a more detailed explanation of the Floating Point Math Package, refer to the FPMP-11 User's Manual DEC-11-NFPMA-A-D.

Examples of the calling conventions are:

```
POLISH MODE:        .
                    .
                    .
                    JSR R4,$POLSH       ;enter Polish mode
                    $subr1              ;call desired subroutines
                    $subr2
                    .
                    .
                    .
                    $subrn              ;call last subroutine desired
                    .WORD    .+2        ;leave Polish mode
                    .
                    .
                    .
                    .
-----------------------------------------------------------------------
J5RR:               .
                    .
                    .
                    .
                    JSR  R5,subr        ;call desired subroutine
                    BR       XX
                    .WORD    arg1       ;subroutine argument address
                    .WORD    arg2
                    .
                    .
                    .
                    .WORD    argn       ;last argument
XX:                 .                   ;return point
                    .
                    .
-----------------------------------------------------------------------
```

```
JPC:              .
                  .
                  .
                  push args onto stack
                  JSR PC,subr
                  .
                  .
                  .
```

H.1 OTS ROUTINES

These are the routines taken from the FORTRAN operating time system. The codes used in the following table are:

S = Routine is included in the standard single precision (2-word) package.
D = Routine is included in the standard double precision (4-word) package.
SD = Routine is included in both standard packages.

Octal codes shown in parentheses are not yet implemented.

NAME	OCTAL CODE	PKG	# OF ARGU	MODE	DESCRIPTION
$ADD	14	D	2	Polish	The double precision add routine. Adds the top stack item (4-words) to the second item (4-words) and leaves the four word sum in their place.
$ADR	12	S	2	Polish	The single precision add routine. Same as $ADD except it uses 2 word numbers.
AINT	26	S	1	J5RR	Returns sign of argument * greatest real integer = absolute value of the argument in R0,R1.
ALOG	53	S	1	J5RR	Calculates natural logarithm of its single argument and returns a two word result in R0,R1.
ALOG10	54	S	1	J5RR	Same as ALOG, except calculates base-10 logarithm.
ATAN	42	S	1	J5RR	Returns the arctangent of its argument in R0,R1.
ATAN2	(43)	S	2	J5RR	Returns ARCTAN (ARG1/ARG2) in R0,R1.
$CMD	16	D	2	Polish	Compares top 4 word items on the stack, flushes the two items, and returns the following condition codes: 4(SP) @SP N=1,Z=0 4(SP) = @SP N=0,Z=1 4(SP) @SP N=0,Z=0

$CMR	17	S	2	Polish	Same as $CMD except it is for 2 word arguments.
COS	37	S	1	J5RR	Single precision version of DCOS.
DATAN	44	D	1	J5RR	Double precision version of ATAN.
DATAN2	(45)	D	2	J5RR	Double precision version of ATAN2.
DBLE	(34)		1	J5RR	Returns in R0-R3 the double precision equivalent of the single precision (two word) argument.
$DCI	(57)	SD	4	JPC	ASCII to double conversion. Calling sequence: Push address of start of ASCII field. Push length of ASCII field in bytes. Push format scale D (from W.D) position of assumed decimal point (see FORTRAN manual). Push P format scale (see FORTRAN manual). JSR PC,$DCI. Returns 4 word result on top of stack.
$DCO	(61)	SD	5	JPC	Double precision to ASCII conversion. Calling sequence: Push address of start of ASCII field. Push length in bytes of ASCII field (W part of W.D) Push D part of W.D position of decimal point). Push P scale. Push 4 word value to be converted, lowest order word first. JSR PC,$DCO.
DCOS	41	D	1	J5RR	Calculates the cosine of its double precision argument and returns the double precision result in R0-R3.
DEXP	52	D	1	J5RR	Calculates the exponential of its double precision argument, and returns the double precision result in R0-R3.
$DI	(11)	SD		Polish	Converts double precision number on the top of the stack to integer. Leaves result on stack.
$DINT	(76)	D	1	Polish	OTS internal function to find the integer part of a double precision number.

DLOG	55	D	1	J5RR	Double precision (4 word) version of ALOG.
DLOG10	56	D	1	J5RR	Double precision (4 word) version of ALOG10.
$DR	(6)		1	Polish	Replaces the double precision item at the top of the stack with its two word, rounded form.
DSIN	40	D	1	J5RR	Calculates the sine of its double precision arg. and returns the double precision result in R0-R3.
DSQRT	47	D	1	J5RR	Calculates the square root of its double precision arg. and returns the double precision result in R0-R3.
$DVD	23	D	2	Polish	The double precision division routine. Divides the second 4-word item on the stack by the top item and leaves the quotient in their place.
$DVI	(24)		2	Polish	The integer division routine. Calculates 2(SP)/@SP and returns the integer quotient on the top of the stack.
$DVR	25	S	2	Polish	The single precision division routine. Same as $DVD, but for 2 word floating point numbers.
$ECO	(62)	SD	5	JPC	Single precision to ASCII conversion according to E format. Same calling sequence as $DCO except that a 2-word value is to be converted.
EXP	51	S	1	J5RR	Single precision version of DEXP. Returns result in R0,R1.
$FCALL	-	S			Internal OTS routine.
$FCO	(64)	SD	5	JPC	Same as $ECO except uses F format conversion.
FLOAT	(32)		1	J5RR	Returns in R0-R1, the real equivalent of its integer argument.
$GCO	(63)	SD	5	JPC	Same as $ECO except uses G format conversion.

$ICI	(65)		2	JPC	ASCII to integer conversion calling sequence: Push address of start of ASCII field. Push length in bytes of ASCII field. JSR PC,$ICI Returns with integer result on top of stack.
$ICO	(67)		3	JPC	Integer to ASCII conversion. Calling sequence: Push address of ASCII field. Push length in bytes of ASCII field. Push integer value to be converted. JSR PC,$ICO Error will return with C bit set on. Ro-R3 destroyed.
IDINT	(31)		1	J5RR	Returns sign of arg * greatest integer <= arg in R0. Arg is double precision.
$ID	(5)	SD	1	Polish	Convert full word argument on the top of the stack to double precision and return result as top 4-words of stack.
IFIX	(35)		1	J5RR	Returns the truncated and fixed real argument in R0.
INT	(30)		1	J5RR	Same as IDINT for single precision args.
$INTR	(27)	S	1	Polish	Same function as AINT, but called in Polish mode with argument and returns result on the stack.
$IR	(4)	SD	1	Polish	Convert full word argument on the top of the stack to single precision and return result as top 2-words of stack.
$MLD	22	D	2	Polish	Double precision multiply. Replaces the top two doubles on the stack with their product.
$MLI	(20)		2	Polish	Integer multiply. Replaces the top 2 integers on the stack with their full word product.
$MLR	21	S	2	Polish	Single precision multiply. Replaces the top two singles on the stack with their product.
$NGD	(3)	SD	2	Polish	Negate the double precision number on the top of the stack.

$NGI	(1)	SD	1	Polish	Negate the integer on the top of the stack.
$NGR	(2)	SD	1	Polish	Negate the single precision number on the top of the stack.
$OCI	(66)		2	JPC	ASCII to octal conversion. Same call as $ICI.
$OCO	(70)		3	JPC	Octal to ASCII conversion. Same call as $ICO.
$POLSH	-	SD	-	-	Called whenever it is desired to enter Polish mode from normal in-line code. It must be called via a JSR R4,$POLSH.
$POPR3	-	D	-	Polish	Internal routine to pop 2-words from the stack and place them into R0,R1.
$POPR4	-	D	-	Polish	Internal routine to pop 4-words from the stack and place them in R0-R3.
$POPR5	-	D	-	Polish	Internal routine to pop 4-words from the stack and place them in registers R0-R3.
$PSHR1	-	SD		Polish	Internal routine to push the contents of R0 onto the stack.
$PSHR2	-	SD	-	Polish	Same as $PSHR1.
$PSHR3	-	SD	-	Polish	Push R0,R1 onto stack.
$PSHR4	-	SD	-	Polish	Push R0-R3 onto stack.
$PSHR5	-	SD	-	Polish	Same as $PSHR4.
$RCI	(60)	SD	4	JPC	ASCII to single precision conversion. Same calling sequence as $DCI. Returns 2-word result on top of stack.
$RD	(7)			Polish	Converts the single precision number on the top of the stack to double precision format. Leaves result on stack.
$RI	(10)	SD		Polish	Converts single precision number on the top of the stack to integer. Leaves result on stack.
$SBD	15	D		Polish	The double precision subtract routine. Subtracts the double precision number on the top of the stack from the second double precision number on the stack and leaves the result on the top of the stack in their place.

$SBR	13	S		Polish	Same as $SBD but for single precision.
SIN	36	S	1	J5RR	Single precision version of DSIN.
SNGL	(33)		1	J5RR	Rounds double precision argument to single precision. Returns result in R0,R1.
SQRT	46	S	1	J5RR	Single precision version of DSQRT.
TANH	50	S	1	J5RR	Single precision hyperbolic tangent function. Returns (EXP(2*ARG)-1/(EXP(2*ARG)+1) in R0,R1.

H.2 NON-OTS ROUTINES

These routines are written especially for FPMP-11 and should not be called directly by the user.

NAME	OCTAL CODE	PKG	DESCRIPTION
$ERR	-	SD	Internal error handler.
$ERRA	-	SD	Similar to $ERR.
$LDR	71	S	Load FLAC, single precision.
$LDD	72	D	Load FLAC, double precision.
$STR	73	S	Store FLAC, single precision.
$STD	74	D	Store FLAC, double precision.
TRAPH	-	SD	The TRAP handler routines and tables.

H.3 ROUTINES ACCESSED VIA TRAP HANDLER

The following is a table of the FPMP-11 routines which can be accessed via TRAPH, the trap handler. Each routine name (entry point) is preceded by its TRAP code number to be used to access it, and followed by a brief description of its operation when called via the TRAP handler. Those entries which are preceded by an asterisk (*) perform operations only on the FLAC, and address no operands. For example, a TRAP call to the single precision square root routine can be coded as follows:

```
             .
             .
             .
        TRAP  46
             .
             .
             .
```

The net effect of the above TRAP instruction is to replace the contents of the FLAC with its square root and then set the condition codes to reflect the result. Note that since the FLAC is implicitly addressed in this instruction, the TRAP call supplies no other address. For such a TRAP call, the addressing mode bits (bits 6 and 7 of the TRAP instruction) are ignored.

All entries not marked by an asterisk require an operand when called. The operand is addressed in one of the four addressing modes explained in section 3.1.1. of the FPMP-11 User's Manual. The addressing mode is specified in bit 6-7 of the TRAP instruction.

("Operand" is the contents of the location addressed in the TRAP call.)

	OCTAL CODE	NAME	DESCRIPTION
	14	$ADD	Double precision addition routine. Adds operand to the FLAC. Assumes 4-word operand.
	12	$ADR	Single precision addition routine. Adds operand to the FLAC. Assumes 2-word operand.
*	26	AINT	Replaces contents of the FLAC by its integer part. SIGN(FLAC) * greatest integer <= |FLAC| . Assumes 2-word argument in FLAC.
*	53	ALOG	Replaces contents of the FLAC by its natural logarithm. Assumes 2-word argument in FLAC.
*	54	ALOG10	Same as ALOG, except calculates base-10 log.
*	42	ATAN	Replaces contents of the FLAC by its arctangent. Assumes 2-word argument in FLAC.
	16	$CMD	Compares operand to the contents of the FLAC, and returns the following condition codes. FLAC<operand, N=1,Z=0 FLAC=operand, N=0,Z=1 FLAC>operand, N=0,Z=0 Assumes 4-word operands.
	17	$CMR	Same as $CMD, but for 2-word operands.
*	37	COS	Same as DCOS, but for 2-word argument.
*	44	DATAN	Same as ATAN, but for 4-word argument.
*	52	DEXP	Replaces the contents of the FLAC by its exponential. Assumes 4-word argument in the FLAC.

*	55	DLOG	Same as ALOG, but for 4-word argument.
*	56	DLOG10	Same as ALOG10, but for 4-word argument.
*	41	DCOS	Replaces the contents of the FLAC by its cosine. Assumes 4-word argument in the FLAC.
*	40	DSIN	Same as DCOS, but calculates sine instead of cosine.
*	47	DSQRT	Replaces the contents of the FLAC by its square root. Assumes 4-word argument in the FLAC.
	23	$DVD	Double precision division routine. Divides the FLAC by the operand and stores the result in the FLAC. Assumes 4-word operands.
	25	$DVR	Same as $DVD, but for 2-word operands.
*	51	EXP	Same as DEXP, but for 2-word argument.
	72	$LDD	Same as $LDR, but assumes 4-word operand.
	71	$LDR	Replaces the contents of the FLAC by the operand. Assumes 2-word operand.
	22	MLD	Double precision multiplication routine. Multiplies the contents of the FLAC by the operand and stores the result in the FLAC. Assumes 4-word operands.
	21	$MLR	Same as $MLD, but for 2-word operands.
	15	$SBD	The double precision subtraction routine. Subtracts the operand from the contents of the FLAC. Assumes a 4-word operand.
	13	$SBR	Same as $SBD, but for 2-word operand.
*	36	SIN	Same as DSIN, but for 2-word argument.
*	46	SQRT	Same as DSQRT, but for 2-word argument.
	73	$STR	Stores the contents of the FLAC into the operand location. The contents of the FLAC are unchanged.
	74	$STD	Same as $STR, but assumes 4-word operand location.
*	50	TANH	Replaces the contents of the FLAC by its hyperbolic tangent. Assumes 2-word argument.

APPENDIX I

TAPE DUPLICATION

Duplication of paper tapes can be accomplished via low- or high-speed I/O devices by toggling (as with the Bootstrap Loader) the following program directly into memory through the Switch Register. (Refer to Section 6.1.1 in Chapter 6 if necessary, for toggling procedure.)

1. Turn on appropriate device switches and place tape in desired reader.

2. Set ENABLE/HALT switch to HALT.

3. Set Switch Register to the desired starting address and press LOAD ADDR.

4. Set Switch Register to each value listed in the CONTENTS column below, lifting the DEP switch after each setting. (Addresses are automatically incremented.) The desired input device (either Low- or High-Speed Reader) and output device (Low- or High-Speed Punch) are specified in the last two words.

ADDRESS	CONTENTS
0	016700
2	000024
4	016701
6	000022
10	005210
12	105710
14	100376
16	105711
20	100376
22	022021
24	111011
26	000764
30	177560 (LSR) or 177550 (HSR)
32	177564 (LSP) or 177554 (HSP)

5. Set Switch Register to starting address specified in 3 above and press LOAD ADDR.

6. Set ENABLE/HALT switch to ENABLE.

7. Press START switch.

TAPE DUPLICATION

NOTE

This program is recommended as a simple way of duplicating the system tapes. However, for extensive tape duplication, the program shown in section 7.8 is recommended.

APPENDIX J

ASSEMBLY AND LINKING INSTRUCTIONS

J.1 SYSTEMS WITHOUT SWITCH REGISTERS

J.1.1 IOX/IOXLPT

IOX/IOXLPT is provided in both source and relocatable object form. Unless modifications are made to the source it is not necessary to assemble the source tapes. The object tape may be linked with the user's object tapes to produce an absolute tape (.LDA).

J.1.1.1 Assembling IOX - IOX consists of three source tapes (-PA1 to -PA3). These tapes are assembled together in sequence with PAL-11S.

J.1.1.2 Assembling IOXLPT - IOXLPT consists of two source tapes (-PA1 to PA2). These tapes are assembled together in sequence with PAL-11S.

J.1.1.3 Linking IOX and IOXLPT - IOX and IOXLPT are linked by LINK-11S with the user's object tapes to produce an absolute tape.

J.1.2 ODT11X

ODT11X is provided in both source and relocatable object form. Unless modifications are made to the source, it is not necessary to assemble the source tape. The object tape may be linked with the user's object tapes to produce an absolute tape.

J.1.2.1 Assembling ODT11X - ODT11X consists of one source tape (-PAL) which is terminated with the following:

```
.EOT
form feed
.END O.ODT
```

When PAL-11S indicates that it has encountered the .EOT, type return so that it will process the .END statement.

J.1.2.2 Linking ODT11X - ODT11X is linked with user object tapes. It is self starting and should be the first object tape input to LINK-11S so that it will be the program started by the Absolute Loader when the program is loaded.

J.1.3 ED-11

The ED-11 source file is available only in RT-11 format on a flexible diskette. The RT-11 MACRO assembler is required to assemble ED-11. The RT-11 linker (LINK) is used to produce the absolute tape.

J.1.3.1 Assembling ED-11 - The RT-11 commands to assemble ED-11 are as follows:

```
.R MACRO
*EDIT11=DX1:EDIT11
```

J.1.3.2 Linking ED-11 - The RT-11 commands to link ED-11 are as follows:

```
.R LINK
*PP:EDIT11/L=EDIT11
```

J.1.4 PAL-11S

The PAL-11S source file is available only in RT-11 format on a flexible diskette. The RT-11 MACRO assembler is required to assemble PAL-11S. The RT11 linker (LINK) or LINK-11S may be used to produce the absolute tape.

J.1.4.1 Assembling PAL-11S - There are three sources which are assembled separately for PAL-11S. One of these, the symbol table source, is available in three versions: 8K, 12K, and 16K. The RT-11 commands to assemble PAL-11S source files are as follows:

```
.R MACRO
*RELMEM=DX1:RELMEM.PAL      Clear Memory Program
*PSYM08=DX1:PSYM08.PAL      8K Symbol Table
*PSYM12=DX1:PSYM12.PAL      12K Symbol Table
*PSYM16=DX1:PSYM16.PAL      16K Symbol Table
*PAL11S=DX1:PAL11S.PAL      Assembler
```

In addition to the above, IOXLPT is used by PAL-11S. The IOXLPT source is also available in RT-11 format on a flexible diskette. The commands to assemble IOXLPT are:

```
.R MACRO
*IOXLPT=DX1:IOXLPT.PAL
```

J.1.4.2 Linking PAL-11S - PAL-11S may be linked with LINK-11S or the RT-11 linker, LINK. The PAL-11S tape actually contains two programs: RELMEM and PAL-11S. RELMEM precedes PAL-11S on the tape.

Using LINK-11S, link PAL-11S as follows:

1. Link RELMEM as a separate program and do not remove the tape from the punch when finished.

2. Link PAL11S.OBJ, IOXLPT.OBJ, and one of the symbol table object tapes (PSYM08.OBJ, PSYM12.OBJ, or PSYM16.OBJ) in that order. The symbol table tape is selected depending on the size of the memory of the computer on which the program is to be executed. If the target computer has 8K words of memory then PSYM08.OBJ, if 12K then PSYM12.OBJ, and if 16K then PSYM16.OBJ. Specify a top address of 57460 for 12K and 77460 for 16K.

   Do not link PAL-11S to run above 16K. The size of the symbol table is fixed, and there is no need to re-link at a higher address even on large systems.

Using RT-11 LINK, link PAL-11S as follows:

1. Link RELMEM as a separate program as shown

```
.R LINK
*RELMEM/L=RELMEM
```

2. Link 8K, 12K, and 16K versions of PAL-11S

```
.R LINK
*PAL08/L/B:204=PAL11S,IOXLPT,PSYM08
*PAL12/L/B:204=PAL11S,IOXLPT,PSYM12
*PAL16/L/B:204=PAL11S,IOXLPT,PSYM16
```

3. Use RT-11 PIP to punch the tapes. Remember not to remove the tape from the punch after punching RELMEM.

```
.R PIP
*PP:=RELMEM.LDA/B
*PP:=PAL08.LDA/B

remove 8K PAL11S.LDA from punch

*PP:=RELMEM.LDA/B
*PP=PAL12.LDA/B

remove 12K PAL11S.LDA from punch

*PP:=RELMEM.LDA/B
*PP:=PAL16.LDA/B
```

J.1.5 LINK-11S

The LINK-11S source file is available only in RT-11 format on a flexible diskette. The RT-11 MACRO assembler is required to assemble LINK-11S. LINK-11S is composed of two components: LINK-11S proper and IOXLPT. See Section N.1.4.1 for instructions on how to assemble IOXLPT using RT-11.

J.1.5.1 Assembling LINK-11S - The RT-11 commands to assemble LINK-11S follow:

```
.R MACRO
*LINK11=DX1:LINK11
```

J.1.5.2 Linking LINK-11S - LINK-11S may be linked with LINK-11S or the RT-11 linker, LINK. There are two object tapes which are linked together to produce LINK-11S: LINK11.OBJ and IOXLPT.OBJ.

Using LINK-11S to link LINK-11S, link the following two tapes in order: LINK11.OBJ and IOXLPT. If versions are desired for systems with more than 8K, specify a top address of 57460 for 12K and 77460 for 16K.

Using RT-11 LINK to link LINK-11S is a two step process because of a difference in philosophy. An initial link is required which produces a link map so that the size of LINK11S can be determined. A final link is then made with the information obtained in the initial link used to compute the bottom address.

The initial link is executed as follows:

```
.R LINK
*,TT:=LINK11,IOXLPT
```

The value displayed for "HIGH LIMIT" is used to compute the bottom address for the final link. Assume for an example that the following was displayed:

```
HIGH LIMIT = 015572
```

Select 37460, 57460, or 77460 depending on whether an 8K, 12K, or 16K top address is desired. The bottom address is computed as follows:

```
         B = T - 4 + 1000

Where:   B = bottom address
         T = top address
         H = high limit

Example: B = 37460-15572+1000
         B = 22666
```

Using the figures in the example above, the final link for an 8K system would be executed as follows:

```
.R LINK
*PP:/B:22666/L,TT:=LINK11,IOXLPT
```

As a check, examine the link map produced and verify that the high limit matches the one used in the calculations above. In the example, the high limit value must be 37460.

J.2 SYSTEMS WITH SWITCH REGISTERS

J.2.1 Assembling PAL-11A

The following procedures are for assembling the PAL-11 Assembler source tapes. An 8K version of the PAL-11A (V007A) Assembler is required, thus also requiring at least an 8K PDP-11 system.

The Assembler consists of two programs. The first program, on tape 1, is a memory clear program and is very short (DEC-11-UPLAA-A-PA1). The second program is the Assembler proper, and consists of eleven ASCII tapes (DEC-11-UPLAA-A-PA2-PA12). They are assembled as follows:

1. Generate a sufficient amount of blank leader tape.

2. Assemble the memory clear program source tape (DEC-11-UPLAA-A-PA1) and assign the binary output to the high-speed punch. For example, PAL-11A's initial dialogue to specify the 2-pass assembly would be:

```
*S  H
*B  H/E
*L
*T
                 (PA1 assembly - 1st pass)
END?
                 (PA1 assembly - 2nd pass)
000000 ERRORS    (No errors - Do not remove
C                the binary tape from the punch.)
```

3. Assemble the rest of the Assembler's source tapes (PA2 - PA12) in numerical sequence.

   Assign the binary output to the high-speed punch. For example, the initial dialogue should be answered as follows:

```
*S  H
*B  H/E
*L
*T
EOF ?            (Enter tape PA2 for 1st pass)
EOF ?            (End of tape PA2, enter PA3)
EOF ?            (End of tape PA3, enter PA4)
EOF ?            (End of tape PA4, enter PA5)
EOF ?            (End of tape PA5, enter PA6)
EOF ?            (End of tape PA6, enter PA7)
EOF ?            (End of tape PA7, enter PA8)
EOF ?            (End of tape PA8, enter PA9)
EOF ?            (End of tape PA9, enter PA10)
EOF ?            (End of tape PA10, enter PA11)
EOF ?            (End of tape PA11, enter PA12)
MAXCL13 = ****** SIMBC = ******    (End of first pass)
END ?
EOF ?            (Enter tape PA2 for 2nd pass)
EOF ?            (End of tape PA2, enter PA3)
EOF ?            (End of tape PA3, enter PA4)
EOF ?            (End of tape PA4, enter PA5)
EOF ?            (End of tape PA5, enter PA6)
EOF ?            (End of tape PA6, enter PA7)
EOF ?            (End of tape PA7, enter PA8)
EOF ?            (End of tape PA8, enter PA9)
EOF ?            (End of tape PA9, enter PA10)
EOF ?            (End of tape PA10, enter PA11)
EOF ?            (End of tape PA11, enter PA12)
```

```
000000 ERRORS    (End of 2nd pass)
C
*S
```

Note that at the end of the first pass there are two undefined symbols: MAXC13 and SIMBC. These undefined symbols are resolved so that there are no errors reported during the second pass.

Be sure that there is sufficient blank trailer tape on the binary output tape before removing the assembled tape from the punch.

Normally, using high-speed paper tape input and output, this process requires about 45 minutes. If a symbol table and listing are requested, there will be about 750 symbols and about 4500 lines of listing.

J.2.2 Assembling ED-11

ED-11 consists of five source tapes (PA1 to PA5) which are assembled together in sequence with 8K PAL-11A.

J.2.3 ODT-11/ODT-11X

In subsequent discussion, reference to ODT applies to both versions. ODT is supplied on both source and absolute binary tapes.

If the program being debugged requires storage where the version of ODT being used is normally loaded, it is necessary to reassemble ODT after changing the starting location.

The source tape of ODT is in three segments, each separated from the next by blank tape. The first segment contains:

```
.=n                (standard location setting statement)
.EOT
```

where n=13026 for ODT-11 or n=12054 for ODT-11X. This statement tells the Assembler to start assembling at address n. To relocate ODT to another starting address, substitute for segment one a source tape consisting of:

```
.=n                (n is the new load address for ODT)
.EOT
```

The .EOT statement tells the Assembler that this is the end of the segment but not the end of the program -- the Assembler will stop and wait for another tape to be placed in the reader.

The second segment of tape contains the ODT source program. This segment is also terminated with .EOT.

The third segment of the tape consists of the statement:

```
.END  O.ODT
```

where .END means "end of program" and O.ODT represents the starting address of the program (see Section 6.2.3).

When relocating ODT, the first segment of the source tape must be changed to reflect the desired load address. The third segment may be changed to .END without a start address. The latter will cause the Loader to halt upon completion of loading.

The segmentation allows the following assembly forms:

1. Assemble alone but at a new address. A new segment one must be generated and assembled with segments two and three.

2. Assemble immediately after the user's program to be debugged. Assemble the tape of the user's program (ending with .EOT) followed by ODT's segment two and either segment three or a new segment three.

3. Assemble inside the program to be debugged. Assemble the first part of the user program (ending with .EOT) followed by ODT's second segment followed by the second part of the user program.

When setting locations before assembling, it must be noted that immediately preceding ODT a minimum internal stack of 40_8 bytes is required for the ODT-11 and 116_8 bytes is required for ODT-11X. Additional room must be allocated for subroutine calls and possible interrupts while ODT is in control. Twelve bytes maximum will be used by ODT proper for subroutine calls and interrupts, giving a minimum safe stack space of 52_8 bytes for ODT-11 or 130_8 bytes for ODT-11X.

Once a new binary tape of ODT has been assembled, load it using the Absolute Loader as explained in Section 6.2.2. Normally, the program to be debugged is loaded before ODT, since ODT will automatically be in control immediately after loading, unless the third segment of ODT's source tape was altered before assembly. As soon as the tape is read in, ODT will print an * on the Teletype to indicate that it is ready for a command.

J.2.4 Assembling IOX/IOXLPT

In subsequent discussion, reference to IOX applies to both versions. IOX is supplied on both source and absolute binary tapes.

If there is more than 4K of core available and it is desired to load IOX (or IOXLPT) in other than the normal location, IOX must be reassembled.

The code

```
.=15100
.EOT
```

appears at the beginning of the first IOX tape (PA1) and contains the starting address. Create a new tape containing the new starting address desired; be sure to allow enough room for 634_{10} words for IOX, 725_{10} for IOXLPT. For example,

```
.=25100
.EOT
```

Use PAL-11A to assemble IOX and substitute the new section of tape for the first part of the old tape (PA1). After the new section is read, insert the IOX tape in the reader so the read head is past the old starting address and .EOT and type the RETURN key to read in the rest of the tape.

Now read in the second tape (PA2). An EOF? message is output at the end of the second tape. Type the RETURN key and the END? message is printed. Put the tapes through for the second pass of the assembler.

IOX (IOXLPT) can also be assembled with a user program if desired. The .=15100 and .EOT lines must be deleted before IOX is assembled with a user program.

IOX can be assembled into the program wherever desired but if it is the first tape read by the assembler, remove it from the reader before typing the RETURN key (after the EOF? message of the second tape. IOX and IOXLPT have a .END code which would cause the assembly pass to end when read). Assembling a user program and IOX together eliminates the need to read in IOX each time the program is run.

J.2.5 Assembling and Linking PAL-11S

PAL-11S consists of two independent programs. The first program is a memory clear program. The second is the assembler. All programs are available as ASCII source tapes, object modules and as a load module.

The memory clear program, MEMCLR, (DEC-11-UPLSA-A-PA1) consists of one ASCII tape. This program should never need to be assembled. The object module may be used when constructing a new load module of PAL-11S.

The assembler consists of three program modules which are assembled separately and then linked together. The first is the main program called PAL-11S. It consists of 13 ASCII tapes (DEC-UPLSA-A-PA2-PA14). The second module is the symbol table, PALSYM, which consists of 2 ASCII tapes (DEC-11-UPLSA-A-PA15-PA16). The third is IOXLPT consisting of 2 ASCII tapes (DEC-11-UPLSA-A-PA17-PA18). Also included is PALSYM, specially created for 12K and 16K, consisting of one tape each (DEC-11-UPLSA-A-PA19-PA20).

If changes are made in any of these modules, that module must be assembled by PAL-11S (V003A) and the new object module can be linked with the other object modules. It should be noted that assembly of these programs will result in:

Program	Pages of Listing (Decimal)	Number of Symbols (Decimal)
PAL-11S	160	756
PALSYM	11	32
IOXLPT	29	191

Also note that there will be two undefined symbols listed at the end of pass 1. These are forward references on direct assignments which get defined properly in pass 2.

An example of the PAL-11S assembly follows:

```
PAL-11S   V003A
*S H
*B H
*L P
*T P/2                           (first pass on PA1)
END ?                            (2nd pass on PA1)
000000 ERRORS                    (End of tape #1 assembly)
                                 (Remove tape from punch)
```

```
PAL-11S    V003A
*S H
*B H
*L P
*T P/2                        (Insert PA2 for 1st pass)
EOF ?                         (End of PA2, insert PA3)
EOF ?                         (End of PA3, insert PA4)
EOF ?                         (End of PA4, insert PA5)
EOF ?                         (End of PA5, insert PA6)
EOF ?                         (End of PA6, insert PA7)
EOF ?                         (End of PA7, insert PA8)
EOF ?                         (End of PA8, insert PA9)
EOF ?                         (End of PA9, insert PA10)
EOF ?                         (End of PA10, insert PA11)
EOF ?                         (End of PA11, insert PA12)
EOF ?                         (End of PA12, insert PA13)
EOF ?                         (End of PA13, insert PA14)
BINCNT=****** SIMBC=******    (End of PA14 and 1st pass)
END ?                         (Insert PA2 for 2nd pass)
EOF ?                         (End of PA2, insert PA3)
EOF ?                         (End of PA3, insert PA4)
EOF ?                         (End of PA4, insert PA5)
EOF ?                         (End of PA5, insert PA6)
EOF ?                         (End of PA6, insert PA7)
EOF ?                         (End of PA7, insert PA8)
EOF ?                         (End of PA8, insert PA9)
EOF ?                         (End of PA9, insert PA10)
EOF ?                         (End of PA10, insert PA11)
EOF ?                         (End of PA11, insert PA12)
EOF ?                         (End of PA12, insert PA13)
EOF ?                         (End of PA13, insert PA14)
000000 ERRORS                 (End of PA14 and 2nd pass)
                              (Remove tape from punch)

PAL-11S    V003A
*S H
*B H
*L P
*T P/2                        (1st pass on PA15)
EOF ?                         (End of PA15, insert PA16)
END ?                         (End of PA16, insert PA15 for 2nd pass)
EOF ?                         (End of PA15, insert PA16)
000000 ERRORS                 (End of 2nd pass)
                              (Remove tape from punch)

PAL-11S    V003A
*S H
*B H
*L P
*T P/2                        (1st pass on PA17)
EOF ?                         (End of PA17, insert PA18)
END ?                         (End of PA18, insert PA17 for 2nd pass)
EOF ?                         (End of PA17, insert PA18)
000000 ERRORS                 (End of 2nd pass)
                              (Remove tape from punch)

PAL-11S    V003A
*S H
*B H
*L P
*T P/2                        (Pass 1 on PA20)
END ?                         (Pass 2 on PA20)
000000 ERRORS                 (End of pass 2)
                              (Remove tape from punch)
```

ASSEMBLY AND LINKING INSTRUCTIONS

The final load module is constructed by LINK-11S. First the memory clear program object module is processed by the linker and the resulting load module is left in the punch while the PAL-11S, PALSYM, and IOXLPT object modules are linked to create a second load module. The resulting tape contains two load modules. The first clears memory and then jumps to the absolute loader to load the second.

In order to take advantage of core sizes larger than 8K, PALSYM, the symbol table, specially created for 12K core and 16K core, and the object modules are included with the assembler. To link for 12K (or 16K), simply substitute the appropriate object tape for PALSYM (use DEC-11-UPLSA-A-PR5 for 12K or DEC-11-UPLSA-A-PR6 for 16K) specify a top address to LINK-11S of 57460 for 12K (77460 for 16K) and link as described in the preceding paragraph.

Do not relink PAL-11S to run above 16K. The size of the symbol table is fixed, and there is no need to re-link at a higher address even on large systems.

The supplied tapes are identified as follows:

Library Code			Contents
DEC-11-UPLSA-A-PA1	Tape 1 of 20	One Assembly	RELMEM (Memory Clear Program)
DEC-11-UPLSA-A-PA2	Tape 2 of 20	One Assembly	PAL-11S (Main Program)
DEC-11-UPLSA-A-PA3	Tape 3 of 20		
DEC-11-UPLSA-A-PA4	Tape 4 of 20		
DEC-11-UPLSA-A-PA5	Tape 5 of 20		
DEC-11-UPLSA-A-PA6	Tape 6 of 20		
DEC-11-UPLSA-A-PA7	Tape 7 of 20		
DEC-11-UPLSA-A-PA8	Tape 8 of 20		
DEC-11-UPLSA-A-PA9	Tape 9 of 20		
DEC-11-UPLSA-A-PA10	Tape 10 of 20		
DEC-11-UPLSA-A-PA11	Tape 11 of 20		
DEC-11-UPLSA-A-PA12	Tape 12 of 20		
DEC-11-UPLSA-A-PA13	Tape 13 of 20		
DEC-11-UPLSA-A-PA14	Tape 14 of 20		
DEC-11-UPLSA-A-PA15	Tape 15 of 20	One Assembly	PALSYM (Symbol Table) for
DEC-11-UPLSA-A-PA16	Tape 16 of 20		
DEC11-UPLSA-A-PA17	Tape 17 of 20	One Assembly	
DEC-11-UPLSA-A-PA18	Tape 18 of 20		IOXLPT
DEC-11-UPLSA-A-PA19	Tape 19 of 20	One Assembly	PALSYM (Symbol Table) for
DEC-11-UPLSA-A-PA20	Tape 20 of 20	One Assembly	PALSYM (Symbol Table) for
DEC-11-UPLSA-A-PR1	Tape 1 of 6		RELMEM Object Module
DEC-11-UPLSA-A-PR2	Tape 2 of 6		PAL-11S Object Module
DEC-11-UPLSA-A-PR3	Tape 3 of 6		PALSYM Object Module for
DEC-11-UPLSA-A-PR4	Tape 4 of 6		IOXLPT Object Module
DEC-11-UPLSA-A-PR5	Tape 5 of 6		PALSYM Object Module for assembler
DEC-11-UPLSA-A-PR6	Tape 6 of 6		PALSYM Object Module for Assembler
DEC-11-UPLSA-A-PL			PAL-11S Load Module[1]

[1] This tape is the concatenation of a link of the RELMEM object module followed by a link of the PAL-11S, PALSYM for 8K, and IOXLPT object modules.

J.2.6 Assembling And Linking LINK-11S

LINK-11S is available as an absolute load module (for an 8K machine), as two object modules (for relinking) and as several ASCII source tapes. There is one object module for the Linker and one for IOXLPT. The supplied object modules may be relinked (using the supplied load module) to load into any size machine larger than 8K. However, the resulting Linker will still assume a top of memory corresponding to an 8K machine (this can be overridden in the command string options). The assumed top of memory and reserved Absolute Loader space may be changed by editing the first linker ASCII tape with ED-11. The parameters to be changed are HGHMEM (high memory address +1 (always even)) and ALODSZ (Absolute Loader size (always even)). The source tapes for the Linker may then be assembled with PAL-11S and the new object module can then replace the supplied Linker object module.

The tapes are identified as follows:

Library Code			
DEC-11-ULKSA-A-PA1	Tape 1 of 6	One Assembly	LINK-11S (Main Program)
DEC-11-ULKSA-A-PA2	Tape 2 of 6		
DEC-11-ULKSA-A-PA3	Tape 3 of 6		
DEC-11-ULKSA-A-PA4	Tape 4 of 6		
DEC-11-ULKSA-A-PA5	Tape 5 of 6	One Assembly	IOXLPT
DEC-11-ULKSA-A-PA6	Tape 6 of 6		
DEC-11-ULKSA-A-PR1	Tape 1 of 2		LINK-11S Object Module
DEC-11-ULKSA-A-PR2	Tape 2 of 2		IOXLPT Object Module
DEC-11-ULKSA-A-PL			LINK-11S Load Module

APPENDIX K

STANDARD PDP-11 ABBREVIATIONS

Abbreviation	Definition
ABS	absolute
A/D	analog-to-digital
ADC	add carry
ADRS	address
ASCII	American Standard Code for Information Interchange
ASL	arithmetic shift left
ASR	arithmetic shift right automatic send/receive
B	byte
BAR	bus address register
BBSY	bus busy
BCC	branch if carry clear
BCS	branch if carry set
BEQ	branch if equal
BG	bus grant
BGE	branch if greater or equal
BGT	branch if greater than
BHI	branch if higher
BHIS	branch if higher or same
BIC	bit clear
BIS	bit set
BIT	bit test
BLE	branch if less or equal
BLOS	branch if lower or same
BLT	branch if less than
BMI	branch if minus
BNE	branch if not equal
BPL	branch if plus
BR	branch
BRD	bus register data
BRX	bus request
BSP	back space
BSR	bus shift register back space record
BSY	busy
BVC	branch if overflow clear
BVS	branch if overflow set
CBR	console bus request
CLC	clear carry
CLK	clock
CLN	clear negative
CLR	clear
CLV	clear overflow

STANDARD PDP-11 ABBREVIATIONS

CLZ	clear zero
CMP	compare
CNPR	console nonprocessor request
CNTL	control
COM	complement
COND	condition
CONS	console
CONT	contents
	continue
CP	central processor
CSR	control and status register
D	data
D/A	digital-to-analog
DAR	device address register
DATI	data in
DATIP	data in, pause
DATO	data out
DATOB	data out, byte
DBR	data buffer register
DCDR	decoder
DE	destination effective address
DEC	decrement
	Digital Equipment Corp.
DEL	delay
DEP	deposit
DEPF	deposit flag
DIV	divide
DMA	direct memory access
DSEL	device select
DST	destination
DSX	display, X-deflection register
EMT	emulator trap
ENB	enable
EOF	end-of-file
EOM	end-of-medium
ERR	error
EX	external
EXAM	examine
EXAMF	examine flag
EXEC	execute
EXR	external reset
F	flag (part of signal name)
FCTN	function
FILO	first in,last out
FLG	flag
GEN	generator
INDIVR	integer divide routine
INC	increment
	increase
INCF	increment flag
IND	indicator
INH	inhibit
INIT	initialize
INST	instruction
INTR	interrupt
INTRF	interrupt flag
I/O	input/output
IOT	input/output trap
IOX	input/output executive routine

STANDARD PDP-11 ABBREVIATIONS

IR	instruction register
IRD	instruction register decoder
ISR	instruction shift register
JMP	jump
JSR	jump to subroutine
LIFO	last in,first out
LKS	line time clock status register
LOC	location
LP	line printer
LSB	least significant bit
LSBY	least significant byte
LSD	least significant digit
MA	memory address
MAR	memory address register
MBR	memory buffer register
MEM	memory
ML	memory location
MOV	move
MSB	most significant bit
MSBY	most significant byte
MSD	most significant digit
MSEL	memory select
MSYN	master sync
ND	negative driver
NEG	negate
NOR	normalize
NPG	nonprocessor grant
NPR	nonprocessor request
NPRF	nonprocessor request flag
NS	negative switch
ODT	octal debugging technique
OP	operate
	operation
OPR	operator
	operand
PA	parity available
PAL	program assembly language
PB	parity bit
PC	program counter
PD	positive driver
PDP	programmed data processor
PERIF	peripheral
PGM	program
PP	paper tape punch
PPB	paper tape punch buffer register
PPS	paper tape punch status register
PR	paper tape reader
PRB	paper tape reader buffer register
PROC	processor
PRS	paper tape reader status register
PS	processor status
	positive switch
PTR	priority transfer
PTS	paper tape software system
PUN	punch

STANDARD PDP-11 ABBREVIATIONS

RD	read
RDR	reader
REG	register
REL	release
RES	reset
ROL	rotate left
ROM	read-only memory
ROR	rotate right
R/S	rotate shift
RTI	return from interrupt
RTS	return from subroutine
R/W	read/write
R/WSR	read/write shift register
S	single
SACK	selection acknowledge
SBC	SUBTRACT CARRY
SC	single cycle
SE	source effective address
SEC	set carry
SEL	select
SEN	set negative
SEV	set overflow
SEX	sign extend
SEZ	set zero
SI	single instruction
SP	stack pointer
	spare
SR	switch register
SRC	source
SSYN	slave sync
ST	start
STPM	set trap marker
STR	strobe
SUB	subtract
SVC	service
SWAB	swap byte
TA	trap address
	track address
TEMP	temporary
TK	teletype keyboard
TKB	teletype keyboard buffer register
TKS	teletype keyboard status register
TP	teletype printer
TPS	teletype printer status register
TRT	trace trap
TSC	timing state control
TST	test
UTR	user trap
VEC	vector
WC	word count
WCR	word count register
XDR	X-line driver
XRCG	X-line read control group
XWCG	X-line write control group
YDR	Y-line driver
YRCG	Y-line read control group
YWCG	Y-line write control group

APPENDIX L

CONVERSION TABLES

L.1 OCTAL-DECIMAL INTEGER CONVERSIONS

0000 to 0777 (Octal) | 0000 to 0511 (Decimal)

Octal	Decimal
10000	4096
20000	8192
30000	12288
40000	16384
50000	20480
60000	24576
70000	28672

	0	1	2	3	4	5	6	7
0000	0000	0001	0002	0003	0004	0005	0006	0007
0010	0008	0009	0010	0011	0012	0013	0014	0015
0020	0016	0017	0018	0019	0020	0021	0022	0023
0030	0024	0025	0026	0027	0028	0029	0030	0031
0040	0032	0033	0034	0035	0036	0037	0038	0039
0050	0040	0041	0042	0043	0044	0045	0046	0047
0060	0048	0049	0050	0051	0052	0053	0054	0055
0070	0056	0057	0058	0059	0060	0061	0062	0063
0100	0064	0065	0066	0067	0068	0069	0070	0071
0110	0072	0073	0074	0075	0076	0077	0078	0079
0120	0080	0081	0082	0083	0084	0085	0086	0087
0130	0088	0089	0090	0091	0092	0093	0094	0095
0140	0096	0097	0098	0099	0100	0101	0102	0103
0150	0104	0105	0106	0107	0108	0109	0110	0111
0160	0112	0113	0114	0115	0116	0117	0118	0119
0170	0120	0121	0122	0123	0124	0125	0126	0127
0200	0128	0129	0130	0131	0132	0133	0134	0135
0210	0136	0137	0138	0139	0140	0141	0142	0143
0220	0144	0145	0146	0147	0148	0149	0150	0151
0230	0152	0153	0154	0155	0156	0157	0158	0159
0240	0160	0161	0162	0163	0164	0165	0166	0167
0250	0168	0169	0170	0171	0172	0173	0174	0175
0260	0176	0177	0178	0179	0180	0181	0182	0183
0270	0184	0185	0186	0187	0188	0189	0190	0191
0300	0192	0193	0194	0195	0196	0197	0198	0199
0310	0200	0201	0202	0203	0204	0205	0206	0207
0320	0208	0209	0210	0211	0212	0213	0214	0215
0330	0216	0217	0218	0219	0220	0221	0222	0223
0340	0224	0225	0226	0227	0228	0229	0230	0231
0350	0232	0233	0234	0235	0236	0237	0238	0239
0360	0240	0241	0242	0243	0244	0245	0246	0247
0370	0248	0249	0250	0251	0252	0253	0254	0255

	0	1	2	3	4	5	6	7
0400	0256	0257	0258	0259	0260	0261	0262	0263
0410	0264	0265	0266	0267	0268	0269	0270	0271
0420	0272	0273	0274	0275	0276	0277	0278	0279
0430	0280	0281	0282	0283	0284	0285	0286	0287
0440	0288	0289	0290	0291	0292	0293	0294	0295
0450	0296	0297	0298	0299	0300	0301	0302	0303
0460	0304	0305	0306	0307	0308	0309	0310	0311
0470	0312	0313	0314	0315	0316	0317	0318	0319
0500	0320	0321	0322	0323	0324	0325	0326	0327
0510	0328	0329	0330	0331	0332	0333	0334	0335
0520	0336	0337	0338	0339	0340	0341	0342	0343
0530	0344	0345	0346	0347	0348	0349	0350	0351
0540	0352	0353	0354	0355	0356	0357	0358	0359
0550	0360	0361	0362	0363	0364	0365	0366	0367
0560	0368	0369	0370	0371	0372	0373	0374	0375
0570	0376	0377	0378	0379	0380	0381	0382	0383
0600	0384	0385	0386	0387	0388	0389	0390	0391
0610	0392	0393	0394	0395	0396	0397	0398	0399
0620	0400	0401	0402	0403	0404	0405	0406	0407
0630	0408	0409	0410	0411	0412	0413	0414	0415
0640	0416	0417	0418	0419	0420	0421	0422	0423
0650	0424	0425	0426	0427	0428	0429	0430	0431
0660	0432	0433	0434	0435	0436	0437	0438	0439
0670	0440	0441	0442	0443	0444	0445	0446	0447
0700	0448	0449	0450	0451	0452	0453	0454	0455
0710	0456	0457	0458	0459	0460	0461	0462	0463
0720	0464	0465	0466	0467	0468	0469	0470	0471
0730	0472	0473	0474	0475	0476	0477	0478	0479
0740	0480	0481	0482	0483	0484	0485	0486	0487
0750	0488	0489	0490	0491	0492	0493	0494	0495
0760	0496	0497	0498	0499	0500	0501	0502	0503
0770	0504	0505	0506	0507	0508	0509	0510	0511

1000 to 1777 (Octal) | 0512 to 1023 (Decimal)

	0	1	2	3	4	5	6	7
1000	0512	0513	0514	0515	0516	0517	0518	0519
1010	0520	0521	0522	0523	0524	0525	0526	0527
1020	0528	0529	0530	0531	0532	0533	0534	0535
1030	0536	0537	0538	0539	0540	0541	0542	0543
1040	0544	0545	0546	0547	0548	0549	0550	0551
1050	0552	0553	0554	0555	0556	0557	0558	0559
1060	0560	0561	0562	0563	0564	0565	0566	0567
1070	0568	0569	0570	0571	0572	0573	0574	0575
1100	0576	0577	0578	0579	0580	0581	0582	0583
1110	0584	0585	0586	0587	0588	0589	0590	0591
1120	0592	0593	0594	0595	0596	0597	0598	0599
1130	0600	0601	0602	0603	0604	0605	0606	0607
1140	0608	0609	0610	0611	0612	0613	0614	0615
1150	0616	0617	0618	0619	0620	0621	0622	0623
1160	0624	0625	0626	0627	0628	0629	0630	0631
1170	0632	0633	0634	0635	0636	0637	0638	0639
1200	0640	0641	0642	0643	0644	0645	0646	0647
1210	0648	0649	0650	0651	0652	0653	0654	0655
1220	0656	0657	0658	0659	0660	0661	0662	0663
1230	0664	0665	0666	0667	0668	0669	0670	0671
1240	0672	0673	0674	0675	0676	0677	0678	0679
1250	0680	0681	0682	0683	0684	0685	0686	0687
1260	0688	0689	0690	0691	0692	0693	0694	0695
1270	0696	0697	0698	0699	0700	0701	0702	0703
1300	0704	0705	0706	0707	0708	0709	0710	0711
1310	0712	0713	0714	0715	0716	0717	0718	0719
1320	0720	0721	0722	0723	0724	0725	0726	0727
1330	0728	0729	0730	0731	0732	0733	0734	0735
1340	0736	0737	0738	0739	0740	0741	0742	0743
1350	0744	0745	0746	0747	0748	0749	0750	0751
1360	0752	0753	0754	0755	0756	0757	0758	0759
1370	0760	0761	0762	0763	0764	0765	0766	0767

	0	1	2	3	4	5	6	7
1400	0768	0769	0770	0771	0772	0773	0774	0775
1410	0776	0777	0778	0779	0780	0781	0782	0783
1420	0784	0785	0786	0787	0788	0789	0790	0791
1430	0792	0793	0794	0795	0796	0797	0798	0799
1440	0800	0801	0802	0803	0804	0805	0806	0807
1450	0808	0809	0810	0811	0812	0813	0814	0815
1460	0816	0817	0818	0819	0820	0821	0822	0823
1470	0824	0825	0826	0827	0828	0829	0830	0831
1500	0832	0833	0834	0835	0836	0837	0838	0839
1510	0840	0841	0842	0843	0844	0845	0846	0847
1520	0848	0849	0850	0851	0852	0853	0854	0855
1530	0856	0857	0858	0859	0860	0861	0862	0863
1540	0864	0865	0866	0867	0868	0869	0870	0871
1550	0872	0873	0874	0875	0876	0877	0878	0879
1560	0880	0881	0882	0883	0884	0885	0886	0887
1570	0888	0889	0890	0891	0892	0893	0894	0895
1600	0896	0897	0898	0899	0900	0901	0902	0903
1610	0904	0905	0906	0907	0908	0909	0910	0911
1620	0912	0913	0914	0915	0916	0917	0918	0919
1630	0920	0921	0922	0923	0924	0925	0926	0927
1640	0928	0929	0930	0931	0932	0933	0934	0935
1650	0936	0937	0938	0939	0940	0941	0942	0943
1660	0944	0945	0946	0947	0948	0949	0950	0951
1670	0952	0953	0954	0955	0956	0957	0958	0959
1700	0960	0961	0962	0963	0964	0965	0966	0967
1710	0968	0969	0970	0971	0972	0973	0974	0975
1720	0976	0977	0978	0979	0980	0981	0982	0983
1730	0984	0985	0986	0987	0988	0989	0990	0991
1740	0992	0993	0994	0995	0996	0997	0998	0999
1750	1000	1001	1002	1003	1004	1005	1006	1007
1760	1008	1009	1010	1011	1012	1013	1014	1015
1770	1016	1017	1018	1019	1020	1021	1022	1023

OCTAL-DECIMAL INTEGER CONVERSIONS (Continued)

2000 to 2777 (Octal) | 1024 to 1535 (Decimal)

Octal	Decimal
10000 -	4096
20000 -	8192
30000 -	12288
40000 -	16384
50000 -	20480
60000 -	24576
70000 -	28672

	0	1	2	3	4	5	6	7
2000	1024	1025	1026	1027	1028	1029	1030	1031
2010	1032	1033	1034	1035	1036	1037	1038	1039
2020	1040	1041	1042	1043	1044	1045	1046	1047
2030	1048	1049	1050	1051	1052	1053	1054	1055
2040	1056	1057	1058	1059	1060	1061	1062	1063
2050	1064	1065	1066	1067	1068	1069	1070	1071
2060	1072	1073	1074	1075	1076	1077	1078	1079
2070	1080	1081	1082	1083	1084	1085	1086	1087
2100	1088	1089	1090	1091	1092	1093	1094	1095
2110	1096	1097	1098	1099	1100	1101	1102	1103
2120	1104	1105	1106	1107	1108	1109	1110	1111
2130	1112	1113	1114	1115	1116	1117	1118	1119
2140	1120	1121	1122	1123	1124	1125	1126	1127
2150	1128	1129	1130	1131	1132	1133	1134	1135
2160	1136	1137	1138	1139	1140	1141	1142	1143
2170	1144	1145	1146	1147	1148	1149	1150	1151
2200	1152	1153	1154	1155	1156	1157	1158	1159
2210	1160	1161	1162	1163	1164	1165	1166	1167
2220	1168	1169	1170	1171	1172	1173	1174	1175
2230	1176	1177	1178	1179	1180	1181	1182	1183
2240	1184	1185	1186	1187	1188	1189	1190	1191
2250	1192	1193	1194	1195	1196	1197	1198	1199
2260	1200	1201	1202	1203	1204	1205	1206	1207
2270	1208	1209	1210	1211	1212	1213	1214	1215
2300	1216	1217	1218	1219	1220	1221	1222	1223
2310	1224	1225	1226	1227	1228	1229	1230	1231
2320	1232	1233	1234	1235	1236	1237	1238	1239
2330	1240	1241	1242	1243	1244	1245	1246	1247
2340	1248	1249	1250	1251	1252	1253	1254	1255
2350	1256	1257	1258	1259	1260	1261	1262	1263
2360	1264	1265	1266	1267	1268	1269	1270	1271
2370	1272	1273	1274	1275	1276	1277	1278	1279

	0	1	2	3	4	5	6	7
2400	1280	1281	1282	1283	1284	1285	1286	1287
2410	1288	1289	1290	1291	1292	1293	1294	1295
2420	1296	1297	1298	1299	1300	1301	1302	1303
2430	1304	1305	1306	1307	1308	1309	1310	1311
2440	1312	1313	1314	1315	1316	1317	1318	1319
2450	1320	1321	1322	1323	1324	1325	1326	1327
2460	1328	1329	1330	1331	1332	1333	1334	1335
2470	1336	1337	1338	1339	1340	1341	1342	1343
2500	1344	1345	1346	1347	1348	1349	1350	1351
2510	1352	1353	1354	1355	1356	1357	1358	1359
2520	1360	1361	1362	1363	1364	1365	1366	1367
2530	1368	1369	1370	1371	1372	1373	1374	1375
2540	1376	1377	1378	1379	1380	1381	1382	1383
2550	1384	1385	1386	1387	1388	1389	1390	1391
2560	1392	1393	1394	1395	1396	1397	1398	1399
2570	1400	1401	1402	1403	1404	1405	1406	1407
2600	1408	1409	1410	1411	1412	1413	1414	1415
2610	1416	1417	1418	1419	1420	1421	1422	1423
2620	1424	1425	1426	1427	1428	1429	1430	1431
2630	1432	1433	1434	1435	1436	1437	1438	1439
2640	1440	1441	1442	1443	1444	1445	1446	1447
2650	1448	1449	1450	1451	1452	1453	1454	1455
2660	1456	1457	1458	1459	1460	1461	1462	1463
2670	1464	1465	1466	1467	1468	1469	1470	1471
2700	1472	1473	1474	1475	1476	1477	1478	1479
2710	1480	1481	1482	1483	1484	1485	1486	1487
2720	1488	1489	1490	1491	1492	1493	1494	1495
2730	1496	1497	1498	1499	1500	1501	1502	1503
2740	1504	1505	1506	1507	1508	1509	1510	1511
2750	1512	1513	1514	1515	1516	1517	1518	1519
2760	1520	1521	1522	1523	1524	1525	1526	1527
2770	1528	1529	1530	1531	1532	1533	1534	1535

3000 to 3777 (Octal) | 1536 to 2047 (Decimal)

	0	1	2	3	4	5	6	7
3000	1536	1537	1538	1539	1540	1541	1542	1543
3010	1544	1545	1546	1547	1548	1549	1550	1551
3020	1552	1553	1554	1555	1556	1557	1558	1559
3030	1560	1561	1562	1563	1564	1565	1566	1567
3040	1568	1569	1570	1571	1572	1573	1574	1575
3050	1576	1577	1578	1579	1580	1581	1582	1583
3060	1584	1585	1586	1587	1588	1589	1590	1591
3070	1592	1593	1594	1595	1596	1597	1598	1599
3100	1600	1601	1602	1603	1604	1605	1606	1607
3110	1608	1609	1610	1611	1612	1613	1614	1615
3120	1616	1617	1618	1619	1620	1621	1622	1623
3130	1624	1625	1626	1627	1628	1629	1630	1631
3140	1632	1633	1634	1635	1636	1637	1638	1639
3150	1640	1641	1642	1643	1644	1645	1646	1647
3160	1648	1649	1650	1651	1652	1653	1654	1655
3170	1656	1657	1658	1659	1660	1661	1662	1663
3200	1664	1665	1666	1667	1668	1669	1670	1671
3210	1672	1673	1674	1675	1676	1677	1678	1679
3220	1680	1681	1682	1683	1684	1685	1686	1687
3230	1688	1689	1690	1691	1692	1693	1694	1695
3240	1696	1697	1698	1699	1700	1701	1702	1703
3250	1704	1705	1706	1707	1708	1709	1710	1711
3260	1712	1713	1714	1715	1716	1717	1718	1719
3270	1720	1721	1722	1723	1724	1725	1726	1727
3300	1728	1729	1730	1731	1732	1733	1734	1735
3310	1736	1737	1738	1739	1740	1741	1742	1743
3320	1744	1745	1746	1747	1748	1749	1750	1751
3330	1752	1753	1754	1755	1756	1757	1758	1759
3340	1760	1761	1762	1763	1764	1765	1766	1767
3350	1768	1769	1770	1771	1772	1773	1774	1775
3360	1776	1777	1778	1779	1780	1781	1782	1783
3370	1784	1785	1786	1787	1788	1789	1790	1791

	0	1	2	3	4	5	6	7
3400	1792	1793	1794	1795	1796	1797	1798	1799
3410	1800	1801	1802	1803	1804	1805	1806	1807
3420	1808	1809	1810	1811	1812	1813	1814	1815
3430	1816	1817	1818	1819	1820	1821	1822	1823
3440	1824	1825	1826	1827	1828	1829	1830	1831
3450	1832	1833	1834	1835	1836	1837	1838	1839
3460	1840	1841	1842	1843	1844	1845	1846	1847
3470	1848	1849	1850	1851	1852	1853	1854	1855
3500	1856	1857	1858	1859	1860	1861	1862	1863
3510	1864	1865	1866	1867	1868	1869	1870	1871
3520	1872	1873	1874	1875	1876	1877	1878	1879
3530	1880	1881	1882	1883	1884	1885	1886	1887
3540	1888	1889	1890	1891	1892	1893	1894	1895
3550	1896	1897	1898	1899	1900	1901	1902	1903
3560	1904	1905	1906	1907	1908	1909	1910	1911
3570	1912	1913	1914	1915	1916	1917	1918	1919
3600	1920	1921	1922	1923	1924	1925	1926	1927
3610	1928	1929	1930	1931	1932	1933	1934	1935
3620	1936	1937	1938	1939	1940	1941	1942	1943
3630	1944	1945	1946	1947	1948	1949	1950	1951
3640	1952	1953	1954	1955	1956	1957	1958	1959
3650	1960	1961	1962	1963	1964	1965	1966	1967
3660	1968	1969	1970	1971	1972	1973	1974	1975
3670	1976	1977	1978	1979	1980	1981	1982	1983
3700	1984	1985	1986	1987	1988	1989	1990	1991
3710	1992	1993	1994	1995	1996	1997	1998	1999
3720	2000	2001	2002	2003	2004	2005	2006	2007
3730	2008	2009	2010	2011	2012	2013	2014	2015
3740	2016	2017	2018	2019	2020	2021	2022	2023
3750	2024	2025	2026	2027	2028	2029	2030	2031
3760	2032	2033	2034	2035	2036	2037	2038	2039
3770	2040	2041	2042	2043	2044	2045	2046	2047

OCTAL-DECIMAL INTEGER CONVERSIONS (Continued)

4000 to 4777 (Octal)	2048 to 2559 (Decimal)

Octal	Decimal
10000	4096
20000	8192
30000	12288
40000	16384
50000	20480
60000	24576
70000	28672

	0	1	2	3	4	5	6	7
4000	2048	2049	2050	2051	2052	2053	2054	2055
4010	2056	2057	2058	2059	2060	2061	2062	2063
4020	2064	2065	2066	2067	2068	2069	2070	2071
4030	2072	2073	2074	2075	2076	2077	2078	2079
4040	2080	2081	2082	2083	2084	2085	2086	2087
4050	2088	2089	2090	2091	2092	2093	2094	2095
4060	2096	2097	2098	2099	2100	2101	2102	2103
4070	2104	2105	2106	2107	2108	2109	2110	2111
4100	2112	2113	2114	2115	2116	2117	2118	2119
4110	2120	2121	2122	2123	2124	2125	2126	2127
4120	2128	2129	2130	2131	2132	2133	2134	2135
4130	2136	2137	2138	2139	2140	2141	2142	2143
4140	2144	2145	2146	2147	2148	2149	2150	2151
4150	2152	2153	2154	2155	2156	2157	2158	2159
4160	2160	2161	2162	2163	2164	2165	2166	2167
4170	2168	2169	2170	2171	2172	2173	2174	2175
4200	2176	2177	2178	2179	2180	2181	2182	2183
4210	2184	2185	2186	2187	2188	2189	2190	2191
4220	2192	2193	2194	2195	2196	2197	2198	2199
4230	2200	2201	2202	2203	2204	2205	2206	2207
4240	2208	2209	2210	2211	2212	2213	2214	2215
4250	2216	2217	2218	2219	2220	2221	2222	2223
4260	2224	2225	2226	2227	2228	2229	2230	2231
4270	2232	2233	2234	2235	2236	2237	2238	2239
4300	2240	2241	2242	2243	2244	2245	2246	2247
4310	2248	2249	2250	2251	2252	2253	2254	2255
4320	2256	2257	2258	2259	2260	2261	2262	2263
4330	2264	2265	2266	2267	2268	2269	2270	2271
4340	2272	2273	2274	2275	2276	2277	2278	2279
4350	2280	2281	2282	2283	2284	2285	2286	2287
4360	2288	2289	2290	2291	2292	2293	2294	2295
4370	2296	2297	2298	2299	2300	2301	2302	2303

	0	1	2	3	4	5	6	7
4400	2304	2305	2306	2307	2308	2309	2310	2311
4410	2312	2313	2314	2315	2316	2317	2318	2319
4420	2320	2321	2322	2323	2324	2325	2326	2327
4430	2328	2329	2330	2331	2332	2333	2334	2335
4440	2336	2337	2338	2339	2340	2341	2342	2343
4450	2344	2345	2346	2347	2348	2349	2350	2351
4460	2352	2353	2354	2355	2356	2357	2358	2359
4470	2360	2361	2362	2363	2364	2365	2366	2367
4500	2368	2369	2370	2371	2372	2373	2374	2375
4510	2376	2377	2378	2379	2380	2381	2382	2383
4520	2384	2385	2386	2387	2388	2389	2390	2391
4530	2392	2393	2394	2395	2396	2397	2398	2399
4540	2400	2401	2402	2403	2404	2405	2406	2407
4550	2408	2409	2410	2411	2412	2413	2414	2415
4560	2416	2417	2418	2419	2420	2421	2422	2423
4570	2424	2425	2426	2427	2428	2429	2430	2431
4600	2432	2433	2434	2435	2436	2437	2438	2439
4610	2440	2441	2442	2443	2444	2445	2446	2447
4620	2448	2449	2450	2451	2452	2453	2454	2455
4630	2456	2457	2458	2459	2460	2461	2462	2463
4640	2464	2465	2466	2467	2468	2469	2470	2471
4650	2472	2473	2474	2475	2476	2477	2478	2479
4660	2480	2481	2482	2483	2484	2485	2486	2487
4670	2488	2489	2490	2491	2492	2493	2494	2495
4700	2496	2497	2498	2499	2500	2501	2502	2503
4710	2504	2505	2506	2507	2508	2509	2510	2511
4720	2512	2513	2514	2515	2516	2517	2518	2519
4730	2520	2521	2522	2523	2524	2525	2526	2527
4740	2528	2529	2530	2531	2532	2533	2534	2535
4750	2536	2537	2538	2539	2540	2541	2542	2543
4760	2544	2545	2546	2547	2548	2549	2550	2551
4770	2552	2553	2554	2555	2556	2557	2558	2559

5000 to 5777 (Octal)	2560 to 3071 (Decimal)

	0	1	2	3	4	5	6	7
5000	2560	2561	2562	2563	2564	2565	2566	2567
5010	2568	2569	2570	2571	2572	2573	2574	2575
5020	2576	2577	2578	2579	2580	2581	2582	2583
5030	2584	2585	2586	2587	2588	2589	2590	2591
5040	2592	2593	2594	2595	2596	2597	2598	2599
5050	2600	2601	2602	2603	2604	2605	2606	2607
5060	2608	2609	2610	2611	2612	2613	2614	2615
5070	2616	2617	2618	2619	2620	2621	2622	2623
5100	2624	2625	2626	2627	2628	2629	2630	2631
5110	2632	2633	2634	2635	2636	2637	2638	2639
5120	2640	2641	2642	2643	2644	2645	2646	2647
5130	2648	2649	2650	2651	2652	2653	2654	2655
5140	2656	2657	2658	2659	2660	2661	2662	2663
5150	2664	2665	2666	2667	2668	2669	2670	2671
5160	2672	2673	2674	2675	2676	2677	2678	2679
5170	2680	2681	2682	2683	2684	2685	2686	2687
5200	2688	2689	2690	2691	2692	2693	2694	2695
5210	2696	2697	2698	2699	2700	2701	2702	2703
5220	2704	2705	2706	2707	2708	2709	2710	2711
5230	2712	2713	2714	2715	2716	2717	2718	2719
5240	2720	2721	2722	2723	2724	2725	2726	2727
5250	2728	2729	2730	2731	2732	2733	2734	2735
5260	2736	2737	2738	2739	2740	2741	2742	2743
5270	2744	2745	2746	2747	2748	2749	2750	2751
5300	2752	2753	2754	2755	2756	2757	2758	2759
5310	2760	2761	2762	2763	2764	2765	2766	2767
5320	2768	2769	2770	2771	2772	2773	2774	2775
5330	2776	2777	2778	2779	2780	2781	2782	2783
5340	2784	2785	2786	2787	2788	2789	2790	2791
5350	2792	2793	2794	2795	2796	2797	2798	2799
5360	2800	2801	2802	2803	2804	2805	2806	2807
5370	2808	2809	2810	2811	2812	2813	2814	2815

	0	1	2	3	4	5	6	7
5400	2816	2817	2818	2819	2820	2821	2822	2823
5410	2824	2825	2826	2827	2828	2829	2830	2831
5420	2832	2833	2834	2835	2836	2837	2838	2839
5430	2840	2841	2842	2843	2844	2845	2846	2847
5440	2848	2849	2850	2851	2852	2853	2854	2855
5450	2856	2857	2858	2859	2860	2861	2862	2863
5460	2864	2865	2866	2867	2868	2869	2870	2871
5470	2872	2873	2874	2875	2876	2877	2878	2879
5500	2880	2881	2882	2883	2884	2885	2886	2887
5510	2888	2889	2890	2891	2892	2893	2894	2895
5520	2896	2897	2898	2899	2900	2901	2902	2903
5530	2904	2905	2906	2907	2908	2909	2910	2911
5540	2912	2913	2914	2915	2916	2917	2918	2919
5550	2920	2921	2922	2923	2924	2925	2926	2927
5560	2928	2929	2930	2931	2932	2933	2934	2935
5570	2936	2937	2938	2939	2940	2941	2942	2943
5600	2944	2945	2946	2947	2948	2949	2950	2951
5610	2952	2953	2954	2955	2956	2957	2958	2959
5620	2960	2961	2962	2963	2964	2965	2966	2967
5630	2968	2969	2970	2971	2972	2973	2974	2975
5640	2976	2977	2978	2979	2980	2981	2982	2983
5650	2984	2985	2986	2987	2988	2989	2990	2991
5660	2992	2993	2994	2995	2996	2997	2998	2999
5670	3000	3001	3002	3003	3004	3005	3006	3007
5700	3008	3009	3010	3011	3012	3013	3014	3015
5710	3016	3017	3018	3019	3020	3021	3022	3023
5720	3024	3025	3026	3027	3028	3029	3030	3031
5730	3032	3033	3034	3035	3036	3037	3038	3039
5740	3040	3041	3042	3043	3044	3045	3046	3047
5750	3048	3049	3050	3051	3052	3053	3054	3055
5760	3056	3057	3058	3059	3060	3061	3062	3063
5770	3064	3065	3066	3067	3068	3069	3070	3071

OCTAL-DECIMAL INTEGER CONVERSIONS (Concluded)

6000 to 6777 (Octal)	3072 to 3583 (Decimal)

Octal	Decimal
10000 -	4096
20000 -	8192
30000 -	12288
40000 -	16384
50000 -	20480
60000 -	24576
70000 -	28672

	0	1	2	3	4	5	6	7
6000	3072	3073	3074	3075	3076	3077	3078	3079
6010	3080	3081	3082	3083	3084	3085	3086	3087
6020	3088	3089	3090	3091	3092	3093	3094	3095
6030	3096	3097	3098	3099	3100	3101	3102	3103
6040	3104	3105	3106	3107	3108	3109	3110	3111
6050	3112	3113	3114	3115	3116	3117	3118	3119
6060	3120	3121	3122	3123	3124	3125	3126	3127
6070	3128	3129	3130	3131	3132	3133	3134	3135
6100	3136	3137	3138	3139	3140	3141	3142	3143
6110	3144	3145	3146	3147	3148	3149	3150	3151
6120	3152	3153	3154	3155	3156	3157	3158	3159
6130	3160	3161	3162	3163	3164	3165	3166	3167
6140	3168	3169	3170	3171	3172	3173	3174	3175
6150	3176	3177	3178	3179	3180	3181	3182	3183
6160	3184	3185	3186	3187	3188	3189	3190	3191
6170	3192	3193	3194	3195	3196	3197	3198	3199
6200	3200	3201	3202	3203	3204	3205	3206	3207
6210	3208	3209	3210	3211	3212	3213	3214	3215
6220	3216	3217	3218	3219	3220	3221	3222	3223
6230	3224	3225	3226	3227	3228	3229	3230	3231
6240	3232	3233	3234	3235	3236	3237	3238	3239
6250	3240	3241	3242	3243	3244	3245	3246	3247
6260	3248	3249	3250	3251	3252	3253	3254	3255
6270	3256	3257	3258	3259	3260	3261	3262	3263
6300	3264	3265	3266	3267	3268	3269	3270	3271
6310	3272	3273	3274	3275	3276	3277	3278	3279
6320	3280	3281	3282	3283	3284	3285	3286	3287
6330	3288	3289	3290	3291	3292	3293	3294	3295
6340	3296	3297	3298	3299	3300	3301	3302	3303
6350	3304	3305	3306	3307	3308	3309	3310	3311
6360	3312	3313	3314	3315	3316	3317	3318	3319
6370	3320	3321	3322	3323	3324	3325	3326	3327

	0	1	2	3	4	5	6	7
6400	3328	3329	3330	3331	3332	3333	3334	3335
6410	3336	3337	3338	3339	3340	3341	3342	3343
6420	3344	3345	3346	3347	3348	3349	3350	3351
6430	3352	3353	3354	3355	3356	3357	3358	3359
6440	3360	3361	3362	3363	3364	3365	3366	3367
6450	3368	3369	3370	3371	3372	3373	3374	3375
6460	3376	3377	3378	3379	3380	3381	3382	3383
6470	3384	3385	3386	3387	3388	3389	3390	3391
6500	3392	3393	3394	3395	3396	3397	3398	3399
6510	3400	3401	3402	3403	3404	3405	3406	3407
6520	3408	3409	3410	3411	3412	3413	3414	3415
6530	3416	3417	3418	3419	3420	3421	3422	3423
6540	3424	3425	3426	3427	3428	3429	3430	3431
6550	3432	3433	3434	3435	3436	3437	3438	3439
6560	3440	3441	3442	3443	3444	3445	3446	3447
6570	3448	3449	3450	3451	3452	3453	3454	3455
6600	3456	3457	3458	3459	3460	3461	3462	3463
6610	3464	3465	3466	3467	3468	3469	3470	3471
6620	3472	3473	3474	3475	3476	3477	3478	3479
6630	3480	3481	3482	3483	3484	3485	3486	3487
6640	3488	3489	3490	3491	3492	3493	3494	3495
6650	3496	3497	3498	3499	3500	3501	3502	3503
6660	3504	3505	3506	3507	3508	3509	3510	3511
6670	3512	3513	3514	3515	3516	3517	3518	3519
6700	3520	3521	3522	3523	3524	3525	3526	3527
6710	3528	3529	3530	3531	3532	3533	3534	3535
6720	3536	3537	3538	3539	3540	3541	3542	3543
6730	3544	3545	3546	3547	3548	3549	3550	3551
6740	3552	3553	3554	3555	3556	3557	3558	3559
6750	3560	3561	3562	3563	3564	3565	3566	3567
6760	3568	3569	3570	3571	3572	3573	3574	3575
6770	3576	3577	3578	3579	3580	3581	3582	3583

7000 to 7777 (Octal)	3584 to 4095 (Decimal)

	0	1	2	3	4	5	6	7
7000	3584	3585	3586	3587	3588	3589	3590	3591
7010	3592	3593	3594	3595	3596	3597	3598	3599
7020	3600	3601	3602	3603	3604	3605	3606	3607
7030	3608	3609	3610	3611	3612	3613	3614	3615
7040	3616	3617	3618	3619	3620	3621	3622	3623
7050	3624	3625	3626	3627	3628	3629	3630	3631
7060	3632	3633	3634	3635	3636	3637	3638	3639
7070	3640	3641	3642	3643	3644	3645	3646	3647
7100	3648	3649	3650	3651	3652	3653	3654	3655
7110	3656	3657	3658	3659	3660	3661	3662	3663
7120	3664	3665	3666	3667	3668	3669	3670	3671
7130	3672	3673	3674	3675	3676	3677	3678	3679
7140	3680	3681	3682	3683	3684	3685	3686	3687
7150	3688	3689	3690	3691	3692	3693	3694	3695
7160	3696	3697	3698	3699	3700	3701	3702	3703
7170	3704	3705	3706	3707	3708	3709	3710	3711
7200	3712	3713	3714	3715	3716	3717	3718	3719
7210	3720	3721	3722	3723	3724	3725	3726	3727
7220	3728	3729	3730	3731	3732	3733	3734	3735
7230	3736	3737	3738	3739	3740	3741	3742	3743
7240	3744	3745	3746	3747	3748	3749	3750	3751
7250	3752	3753	3754	3755	3756	3757	3758	3759
7260	3760	3761	3762	3763	3764	3765	3766	3767
7270	3768	3769	3770	3771	3772	3773	3774	3775
7300	3776	3777	3778	3779	3780	3781	3782	3783
7310	3784	3785	3786	3787	3788	3789	3790	3791
7320	3792	3793	3794	3795	3796	3797	3798	3799
7330	3800	3801	3802	3803	3804	3805	3806	3807
7340	3808	3809	3810	3811	3812	3813	3814	3815
7350	3816	3817	3818	3819	3820	3821	3822	3823
7360	3824	3825	3826	3827	3828	3829	3830	3831
7370	3832	3833	3834	3835	3836	3837	3838	3839

	0	1	2	3	4	5	6	7
7400	3840	3841	3842	3843	3844	3845	3846	3847
7410	3848	3849	3850	3851	3852	3853	3854	3855
7420	3856	3857	3858	3859	3860	3861	3862	3863
7430	3864	3865	3866	3867	3868	3869	3870	3871
7440	3872	3873	3874	3875	3876	3877	3878	3879
7450	3880	3881	3882	3883	3884	3885	3886	3887
7460	3888	3889	3890	3891	3892	3893	3894	3895
7470	3896	3897	3898	3899	3900	3901	3902	3903
7500	3904	3905	3906	3907	3908	3909	3910	3911
7510	3912	3913	3914	3915	3916	3917	3918	3919
7520	3920	3921	3922	3923	3924	3925	3926	3927
7530	3928	3929	3930	3931	3932	3933	3934	3935
7540	3936	3937	3938	3939	3940	3941	3942	3943
7550	3944	3945	3946	3947	3948	3949	3950	3951
7560	3952	3953	3954	3955	3956	3957	3958	3959
7570	3960	3961	3962	3963	3964	3965	3966	3967
7600	3968	3969	3970	3971	3972	3973	3974	3975
7610	3976	3977	3978	3979	3980	3981	3982	3983
7620	3984	3985	3986	3987	3988	3989	3990	3991
7630	3992	3993	3994	3995	3996	3997	3998	3999
7640	4000	4001	4002	4003	4004	4005	4006	4007
7650	4008	4009	4010	4011	4012	4013	4014	4015
7660	4016	4017	4018	4019	4020	4021	1022	4023
7670	4024	4025	4026	4027	4028	4029	4030	4031
7700	4032	4033	4034	4035	4036	4037	4038	4039
7710	4040	4041	4042	4043	4044	4045	4046	4047
7720	4048	4049	4050	4051	4052	4053	4054	4055
7730	4056	4057	4058	4059	4060	4061	4062	4063
7740	4064	4065	4066	4067	4068	4069	4070	4071
7750	4072	4073	4074	4075	4076	4077	4078	4079
7760	4080	4081	4082	4083	4084	4085	4086	4087
7770	4088	4089	4090	4091	4092	4093	4094	4095

L.2 POWERS OF TWO

POWERS OF TWO

2^n	n	n^{-2}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 081 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 634	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 985 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 668 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 834 582 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 171 513 417 041 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 708 520 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 215 395 854 260 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 927 130 126 953 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 963 565 063 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 782 531 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 891 265 869 140 625

L.3 SCALES OF NOTATION

L.3.1 2^x In Decimal

x	2^x	x	2^x	x	2^x
0.001	1.00069 33874 62581	0.01	1.00695 55500 56719	0.1	1.07177 34625 36293
0.002	1.00138 72557 11335	0.02	1.01395 94797 90029	0.2	1.14869 83549 97035
0.003	1.00208 16050 79633	0.03	1.02101 21257 07193	0.3	1.23114 44133 44916
0.004	1.00277 64359 01078	0.04	1.02811 38266 56067	0.4	1.31950 79107 72894
0.005	1.00347 17485 09503	0.05	1.03526 49238 41377	0.5	1.41421 35623 73095
0.006	1.00416 75432 38973	0.06	1.04246 57608 41121	0.6	1.51571 65665 10398
0.007	1.00486 38204 23785	0.07	1.04971 66836 23067	0.7	1.62450 47927 12471
0.008	1.00556 05803 98468	0.08	1.05701 80405 61380	0.8	1.74110 11265 92248
0.009	1.00625 78234 97782	0.09	1.06437 01824 53360	0.9	1.86606 59830 73615

L.3.2 $10^{\pm n}$ In Octal

10^n	n	10^{-n}
1	0	1.000 000 000 000 000 000 00
12	1	0.063 146 314 631 463 146 31
144	2	0.005 075 341 217 270 243 66
1 750	3	0.000 406 111 564 570 651 77
23 420	4	0.000 032 155 613 530 704 15
303 240	5	0.000 002 476 132 610 706 64
3 641 100	6	0.000 000 206 157 364 055 37
46 113 200	7	0.000 000 015 327 745 152 75
575 360 400	8	0.000 000 001 257 143 561 06
7 346 545 000	9	0.000 000 000 104 560 276 41
112 402 762 000	10	0.000 000 000 006 676 337 66
1 351 035 564 000	11	0.000 000 000 000 537 657 77
16 432 451 210 000	12	0.000 000 000 000 043 136 32
221 411 634 520 000	13	0.000 000 000 000 003 411 35
2 657 142 036 440 000	14	0.000 000 000 000 000 264 11
34 327 724 461 500 000	15	0.000 000 000 000 000 022 01
434 157 115 760 200 000	16	0.000 000 000 000 000 001 63
5 432 127 413 542 400 000	17	0.000 000 000 000 000 000 14
67 405 553 164 731 000 000	18	0.000 000 000 000 000 000 01

L.3.3 n Log 2 and 10 In Decimal

n	n $\log_{10} 2$	n $\log_2 10$	n	n $\log_{10} 2$	n $\log_2 10$
1	0.30102 99957	3.32192 80949	6	1.80617 99740	19.93156 85693
2	0.60205 99913	6.64385 61898	7	2.10720 99696	23.25349 66642
3	0.90308 99870	9.96578 42847	8	2.40823 99653	26.57542 47591
4	1.20411 99827	13.28771 23795	9	2.70926 99610	29.89735 28540
5	1.50514 99783	16.60964 04744	10	3.01029 99566	33.21928 09489

L.3.4 Addition and Multiplication, Binary and Octal

Addition **Multiplication**

Binary Scale

0 + 0 = 0
0 + 1 = 1 + 0 = 1
1 + 1 = 10

0 × 0 = 0
0 × 1 = 1 × 0 = 0
1 × 1 = 1

Octal Scale

0	01	02	03	04	05	06	07
1	02	03	04	05	06	07	10
2	03	04	05	06	07	10	11
3	04	05	06	07	10	11	12
4	05	06	07	10	11	12	13
5	06	07	10	11	12	13	14
6	07	10	11	12	13	14	15
7	10	11	12	13	14	15	16

1	02	03	04	05	06	07
2	04	06	10	12	14	16
3	06	11	14	17	22	25
4	10	14	20	24	30	34
5	12	17	24	31	36	43
6	14	22	30	36	44	52
7	16	25	34	43	52	61

L.3.5 Mathematical Constants In Octal

$\pi =$	$3.11037\ 552421_8$	$e =$	$2.55760\ 521305_8$	$\gamma =$	$0.44742\ 147707_8$
$\pi^{-1} =$	$0.24276\ 301556_8$	$e^{-1} =$	$0.27426\ 530661_8$	$\ln \gamma =$	$-\ 0.43127\ 233602_8$
$\sqrt{\pi} =$	$1.61337\ 611067_8$	$\sqrt{e} =$	$1.51411\ 230704_8$	$\log_2 \gamma =$	$-\ 0.62573\ 030645_8$
$\ln \pi =$	$1.11206\ 404435_8$	$\log_{10} e =$	$0.33626\ 754251_8$	$\sqrt{2} =$	$1.32404\ 746320_8$
$\log_2 \pi =$	$1.51544\ 163223_8$	$\log_2 e =$	$1.34252\ 166245_8$	$\ln 2 =$	$0.54271\ 027760_8$
$\sqrt{10} =$	$3.12305\ 407267_8$	$\log_2 10 =$	$3.24464\ 741136_8$	$\ln 10 =$	$2.23273\ 067355_8$

APPENDIX M

NOTE TO USERS OF SERIAL LA30 AND 600, 1200, AND 2400 BAUD VT05'S

The serial LA30 requires that filler characters follow each carriage return; the 600, 1200, and 2400 baud VT05's require that filler characters follow each line feed. The following table lists the filler characters needed. The byte at location 44_8 has been established as the filler count and the byte at location 45_8 contains the character to be filled. These locations are initially set to zero by PAL-11A and ED-11 to allow normal operation of the program.

Depending on the terminal, change the locations as follows:

	LOC 44	LOC 45	Resulting Word (binary)
LA30	011	015	0000110100001001
VT05 600 Baud	001	012	0000101000000001
VT05 1200 Baud	002	012	0000101000000010
VT05 2400 Baud	004	012	0000101000000100

The proper binary word can be stored at location 44 by using the console switches as described in section 2.1.2 of this manual. Furthermore, users with a 2400 baud VT05 should avoid the use of vertical tab characters in their programs. Vertical tabs will not be properly filled and may cause characters to be lost.

Once the changes have been made, the program may be dumped to paper tape by using the bootstrap version of DUMPAB (see section 6.3 in this manual). However, since programs change each time a new version is released, it is necessary to have a program listing to determine the exact memory limits to be dumped.

The above changes only affect output to the console teleprinter.

Users of IOX or IOXLPT source tapes will find the byte at location 44 tagged "I.44:" and the byte at location 45 tagged "I.45:". These locations are defined near the end of the second source tape and can be changed to appropriate values using ED-11.

ODT-11 uses the locations (44 and 45) but does not set them to zero initially.

APPENDIX N

USING THE ABSOLUTE LOADER ON PDP-11'S WITHOUT SWITCH REGISTERS

This appendix describes the procedures for loading and using the Absolute Loader on PDP-11's without switch registers. The procedures are divided into LSI-11, M9301-YB bootstrap loader, and M9301-YA bootstrap loader. Chapter 6 describes the procedures for machines with switch registers.

N.1 LSI-11

The following are instructions for loading and using the Absolute Loader on an LSI-11.

1. Press the BOOT/INIT switch on the LSI-11 front panel to enable the bootstrap loader. An @ prints at the terminal.

2. Place the Absolute Loader tape (DEC-11-UABLB-A-PO) in the reader.

3. Type the status register address of the input device and L to load the Absolute Loader.

   For example, when loading from the console terminal paper tape reader, type:

   ```
   @177560L
   ```

   When the tape has been read, an @ followed by the start address of the Absolute Loader prints at the terminal.

   For example, on a machine with 8K memory, type:

   ```
   @177560L
   ```

   The Absolute Loader prints the address of the Absolute Loader:

   ```
   @37500
   @
   ```

4. Place the tape to be loaded via the Absolute Loader in the reader.

5. Select the type of loading from the following:

   a. Normal Loading

      For normal loading, type the address of the Absolute Loader (printed at the terminal), followed by G, e.g.,

      ```
      @xxx500G
      ```

where xxx is the memory size of the system and is:

xxx	Memory Size
017	4K
037	8K
057	12K
077	16K
117	20K
137	24K
157	28K

For example, in an 8K system, type:

```
@37500G
```

Normal loading can also be achieved by typing the P command, e.g.,

```
@P
```

b. Relocated Loading

Type the software switch register value and deposit the relocation value as follows:

```
@xxx516/yyyyyy zzzzzz↵
@xxx500G
```

or type:

```
@xxx516/yyyyyy zzzzzz↵
@P
```

where xxx516 is dependent on memory size and is the address of the software switch register, yyyyyy is the old content of the switch register, and zzzzzz is the new relocation value.

The value of zzzzzz is explained in Section 6.2.2 for the value of the switch register for relocated loading. For example, in an 8K system, the dialogue would be:

```
@37516/yyyyyy zzzzzz↵
@
```

The following is an example of a normal load on an 8K machine.

```
                ;boot system and put Absolute Loader
                ;in reader
@177560L        ;Absolute Loader tape is read
@37500          ;put tape to be loaded in reader
@P              ;tape is read in.
```

The following is an example of a relocated load on an 8K machine:

```
                     ;boot system
@177560L             ;put Absolute Loader tape
@37500               ;in reader
@37516/000000 1001   ;put tape in reader
@P                   ;tape is read
```

To continue loading, change 1001 in the above example to 1.

6. If more tapes are to be loaded as explained in Section 6.2.2, put the next tape in the reader and repeat section a or b of item 5.

7. If the tape is not self-starting, the halt address of the Absolute Loader is printed, followed by an @. Type the starting address followed by a G to start the program.

```
@37500
@xxxxxxG
```

where xxxxxx is the starting address of the program.

N.2 M9301-YB BOOTSTRAP LOADER

The following are instructions for loading and using the Absolute Loader on a PDP-11 (e.g., PDP-11/04) without a switch register.

1. Press the BOOT/INIT switch on the PDP-11 front panel to enable the bootstrap loader. A $ and four numbers print at the terminal. The four numbers are the values of R0, R4, R6, and the PC, respectively.

   For example:

   ```
   0077400 012450 000546 004054
   $
   ```

2. Place the Absolute Loader (DEC-11-UABLB-A-PO) in the reader.

3. Type the device code (PR for the PC11 high-speed reader or TT for the terminal reader) to load the Absolute Loader.

   ```
   $PR↵
   ```

   or

   ```
   $TT↵
   ```

   when the tape has read in, the machine halts.

4. Place the tape to be loaded by the Absolute Loader in the reader.

5. Select the type of loading from the following:

   a. Normal Loading

      For normal loading, press the CONT switch on the PDP-11 front panel.

   b. Relocated Loading

      1) Press the BOOT/INIT switch; a $ followed by the four numbers explained in item 1 prints at the terminal.

      2) Load the address of the software switch register as follows:

         ```
         $L xxx516↵
         ```

3) Deposit the relocation value in the software switch register as follows:

```
$D yyyyyy↵
```

where yyyyyy is the value explained in Section 6.2.2 for relocated loading.

4) Load the starting address of the Absolute Loader as follows:

```
$L xxx500↵
```

5) Type S to start running the Absolute Loader.

```
$S↵
```

6. If more tapes are to be loaded as explained in Section 6.2.2, put the next tape in the reader and repeat section a or b of item 5.

7. If the tape is not self-starting,

   a. Press the BOOT/INIT switch.

   b. Load the starting address of the program with the L command, i.e.,

   ```
   $L xxxxxx↵
   ```

   c. Start the program with the S command:

   ```
   $S↵
   ```

The following are examples for PDP-11 with 16K words of memory.

Relocated - continuous loading:

```
$L 77516↵

$D 1↵

$L 77500↵

$S↵
```

Relocated - load in specified area of core:

```
$L 77516↵

$D 1001↵

$L 77500↵

$S↵
```

N.3 M9301-YA BOOTSTRAP LOADER

The instructions for loading and using the Absolute Loader on a PDP-11 (e.g., PDP-11/04) without a switch register but with a console terminal are the same as described in Section O.2.

USING THE ABSOLUTE LOADER ON PDP-11'S WITHOUT SWITCH REG

PDP-11's without console terminals may only be loaded with normal loading methods. See the M9301 Maintenance Manual for instructions on placing the address of the paper tape bootstrap in the micro switch on the M9301 module. The following instructions are for PDP-11's without console terminals.

1. Place the Absolute Loader tape (DEC-11-UABLB-A-PO) in the reader.

2. Press the BOOT/INIT switch. When the tape has read in, the machine halts.

3. Place the self-starting tape to be loaded by the Absolute Loader in the reader.

4. Press the CONT switch.

INDEX

Software Handbook
DEC-11-XPTSA-B-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form.

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

____

____

____

____

____

____

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

____

____

____

____

____

____

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

____

____

____

____

____

____

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

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

Name ________________________________ Date ____________________

Organization ____

Street ____

City _____________________ State __________ Zip Code
or
Country __________

If you require a written reply, please check here. □

Please cut along this line.

-- Fold Here --

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

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

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

Postage will be paid by:

digital

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