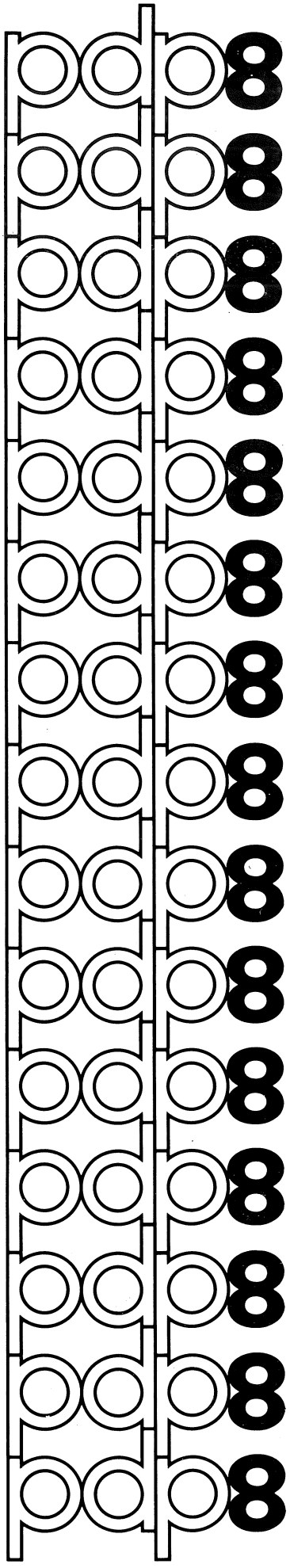
digital



# 4K assemblers

## pal III
## macro-8

digital equipment corporation

DEC-08-LAS4A-A-D

# 4K ASSEMBLERS

# PAL III/MACRO-8

For additional copies, order No. DEC-08-LAS4A-A-D
from Software Distribution Center, Digital Equipment
Corporation, Maynard, Mass.

digital equipment corporation · maynard. massachusetts

First Printing, July 1973

# PREFACE

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

The postage prepaid "READER'S COMMENTS" form on the last page of this document requests the user's critical evaluation. All comments received are acknowledged and will be considered when subsequent documents are prepared.

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

The material in this document is for information purposes only and is subject to change without notice. DIGITAL assumes no responsibility for the use or reliability of software and equipment which is not supplied by it. DIGITAL assumes no responsibility for any errors which may appear in this document.

# contents


**Introduction to 4K Assemblers** ..... 1

**PAL III Programming** ..... 1

Character Set ..... 2
- Legal Characters ..... 2
- Illegal Characters ..... 3

Numbers ..... 3

Format Effectors ..... 3
- Form Feed ..... 3
- Tabulations ..... 3
- Statement Terminators ..... 4

Statements ..... 5
- Labels ..... 5
- Instructions ..... 6
- Operands ..... 6
- Comments ..... 6

Coding Practices ..... 6

Symbols ..... 7
- Internal Symbol Representation for PAL III ..... 7
- Symbolic Addresses ..... 8
- Symbolic Instructions ..... 10
- Symbolic Operands ..... 10
- Symbol Table ..... 11
- Direct Assignment Statements ..... 11

Expressions ..... 14

Address Assignments ..... 17
- Current Address Indicator ..... 17
- Indirect Addressing ..... 18
- Autoindexing ..... 19

Instructions ..... 19
Memory Reference Instructions ..... 20
Microinstructions ..... 20
Pseudo-Operators ..... 23
Program Preparation and Assembler Output ..... 29
Operating Procedures ..... 32
Summary of Diagnostic Messages for PAL III ..... 34

**MACRO-8 Programming** ..... 37
Characters ..... 37
Expressions ..... 37
Origin Setting ..... 39
Link Generation ..... 41
Literals ..... 42
Field Pseudo-Op ..... 46

Text Facility ..... 47
Single Character Text Facility ..... 47
Text Strings ..... 48
Numbers ..... 49

User Defined Macros ..... 53
Defining a Macro ..... 53

MACRO-8 Pseudo-Operators ..... 57

Symbol Table ..... 57
Symbol Table Modification ..... 58
Internal Symbol Representation for MACRO-8 ..... 58
Memory Reference Instruction Recognition ..... 61
Compatibility Between PAL III and MACRO-8 ..... 61

Programming Hints ..... 62
Dealing with a Limited Symbol Space ..... 62

Summary of MACRO-8 Error Diagnostics ..... 66

Pass 3 Output—Assembly Listing ..... 68

MACRO-8 Operating Procedures ..... 69
Assembler Output ..... 69

Operating Instructions ..... 70
MACRO-8 Switch Options ..... 71

**Appendix A** ..... A-1
**Appendix B** ..... B-1
**Appendix C** ..... C-1

# 4K assemblers

## INTRODUCTION TO 4K ASSEMBLERS

This manual contains descriptions of two PDP-8 4K Assemblers, the first and most basic of which is PAL III. It is assumed that the reader is already familiar with the material presented in the first five chapters of *Introduction to Programming,* as the PAL III programming language is discussed in detail in that section.

In addition to PAL III, the MACRO-8 Assembler is also discussed. MACRO-8 is similar to PAL III with the following additional features: user defined macros, double precision integers, floating-point constants, arithmetic and Boolean operators, literals, text facilities, and automatic off-page linkage generation. MACRO-8 is recommended when any of these features is desired and when a large symbol table is not required.

Appendix C contains a list of the permanent symbols for each of these assemblers.

Several other System Library Programs are useful in assembly language programming. The Symbolic Tape Editor can be used to change, correct, or create a program at the Teletype, and after assembly, DDT or ODT are useful in debugging the program. More on these and other useful programs can be found in *Introduction to Programming.*

## PAL III PROGRAMMING

PAL III (an acronym for Program Assembly Language, version III) is a two pass Assembler (with an optional third pass) designed for the 4K PDP-8 family of computers. A program, written in the

PAL III source language, is translated by the Assembler into a binary tape in two passes through the Assembler. The binary tape is loaded by the Binary Loader into the computer for execution.

During the first pass of the assembly, all user symbols are defined and placed in the Assembler's symbol table. During the second pass, the binary equivalents of the input source language are generated and punched. The Assembler's third pass produces a printed assembly listing (a listing of the program's instructions with the location, generated binary, and source code side by side on each line).

The Assembler requires a basic PDP-8 family computer with a 4K core memory, and a Teletype console. The Assembler can also use either the high-speed reader, the high-speed punch, or both. The user can change the Assembler's permanent symbol table to reflect his specific machine configuration, as explained under the section on Altering the Permanent Symbol Table.

**Character Set**

LEGAL CHARACTERS

The following characters are acceptable to PAL III:

1. The alphabetic characters: A through Z
2. The numeric characters: 0 through 9
3. The following special characters:
   a. Printing characters

| | | | |
|---|---|---|---|
| + | plus | ; | semicolon |
| — | minus | $ | dollar sign |
| , | comma | . | period |
| = | equal sign | / | slash |
| * | asterisk | | |

   b. Nonprinting keyboard characters:
      SPACE
      TAB
      RETURN
4. Ignored characters:
   FORM FEED
   blank tape
   RUBOUT
   Leader Trailer (code 200)
   LINE FEED

ILLEGAL CHARACTERS

All other characters are illegal (except when used in a comment) and cause the illegal character message:

IC xxxx AT nnnn

during pass 1, where xxxx is the octal value of the offending character and nnnn is the value of the current location counter where it occurred. Illegal characters are ignored and assembly can proceed. The current location counter contains the address in which the next word of object code will be assembled. If the illegal character occurs in the middle of a symbol, the symbol is terminated at that point.

**Numbers**

Any sequence of digits delimited by punctuation characters forms a number. For example:

1
35
4372

The pseudo-ops OCTAL and DECIMAL indicate to the Assembler which radix, or base, is to be used in number interpretation. The radix is initially set to octal (base 8) and remains octal unless it is changed. It can be changed to decimal (base 10) via the DECIMAL pseudo-op, which indicates that all numbers which follow are to be interpreted as decimal. This is then the case until the occurrence of the OCTAL pseudo-op which converts the base back to octal, and so on. (For an explanation of the internal representation of numbers in the PDP-8, see *Introduction to Programming.*)

**Format Effectors**

FORM FEED

The form feed code, if present in a PAL III program, will cause the Assembler to output 12 blank lines during the pass 3 Assembly listing. This is useful in creating a page-by-page listing. The form feed is generated by typing a SHIFT/L on the Teletype.

TABULATIONS

Tabulations are used in the body of a source program to provide a neat readable listing. Tabs separate fields into columns (for details see Chapter 7 in *Introduction to Programming*). For example, a line written:

```
GO,TAD TOTAL/MAIN LOOP
```

is much easier to read if tabs are inserted to form:

```
GO,     TAD TOTAL       /MAIN LOOP
```

## STATEMENT TERMINATORS

Either the semicolon (;) or the RETURN key may be used as a statement terminator. The semicolon is considered identical to a carriage return/line feed except that it will not terminate a comment. For example:

```
TAD A           /THIS IS A COMMENT;     TAD B
```

The entire expression between the slash (/) and the carriage return is considered a comment. Therefore the Assembler ignores the TAD B.

If, for example, the user wishes to write a sequence of instructions to rotate the contents of the accumulator and link six places to the right, it might look like the following:

```
RTR
RTR
RTR
```

However, the programmer can alternatively place all three instructions on a single line by separating them with the special character semicolon (;) and terminating the line with a carriage return. The above sequence of instructions can then be written:

```
RTR;RTR;RTR
```

These multi-statement lines are particularly useful when setting aside a section of data storage for use during processing. For example, a 4-word cleared block could be reserved by specifying either of the following formats:

```
LIST,   0;      0;      0;      0
```

or

```
LIST,   Ø
        Ø
        Ø
        Ø
```

Either format may be used to input data words which may be in the form of numbers, symbols, or expressions. (Symbols and expressions will be explained later.) Each of the following lines generate one storage word in the object program:

```
DATA,   7777
        A+C-B
        S
        123+B2
```

**Statements**

PAL III source programs are usually prepared on a Teletype, with the aid of the Symbolic Tape Editor, as a sequence of statements. Each statement is written on a single line and is terminated by typing the RETURN key. PAL III statements are virtually format free; that is, elements of a statement are not placed in numbered columns with rigidly controlled spacing, as in punched-card oriented assemblers.

There are four types of elements in a PAL III statement which are identified by the order of their appearance in the statement, and by the separating (or delimiting) character which follows or precedes the element.

Statements are written in the general form:

label,  instruction  operand  /comment

A statement must contain at least one of these elements and may contain all four types. The Assembler interprets and processes the statements, generating one or more binary instructions or data words, or performing an assembly process.

LABELS

A label is the symbolic name created by the programmer to identify the location of a statement in the program. If present, the label is written first in a statement and is terminated by a comma. There must be no intervening spaces between any of the characters and the comma.

INSTRUCTIONS

An instruction may be one or more of the mnemonic machine instructions (see Appendix C), or a pseudo-operation (pseudo-op) which directs assembly processing. (The assembly pseudo-ops are described later in this manual.) Instructions are terminated with one or more spaces (or tabs if an operand follows) or with a semicolon, slash, or carriage return.

OPERANDS

Operands are usually the octal or symbolic addresses of the data to be accessed when an instruction is executed, but they can be any expression, or an argument of a pseudo-op. In each case, interpretation of operands in a statement depends on the statement instruction. Operands are terminated by a semicolon, slash, or carriage return.

COMMENTS

Following a slash mark the programmer may add notes to a statement. Such comments do not affect assembly processing or program execution, but are useful in the program listing for later analysis or debugging. The Assembler ignores everything from the slash to the next carriage return. (For an example see the section on Statement Terminators, preceding.)

It is possible to have nothing but a carriage return on a line, resulting in a space in the final listing. An error message is not given.

**Coding Practices**

A neat printout (or program listing, as it is usually called) makes subsequent editing, debugging, and interpretation much easier than if the coding were laid out in a haphazard fashion. The coding practices listed below are in general use, and will result in a readable, orderly listing.

1. A title comment begins with a slash at the left hand margin.
2. Pseudo-ops may begin at the left margin; often, however, they are indented one tab stop to line up with the executable instructions.
3. Address labels begin at the left margin. They are separated from succeeding fields by a tabulation.

4. Instructions, whether or not they are preceded by a label field, are indented one tab stop.
5. A comment is separated from the preceding field by one or two tabs (as required) and a slash; if the comment occupies the whole line, it usually begins with a slash at the left margin.

**Symbols**

A symbol is a string of letters and digits beginning with a letter and delimited by a non-alphanumeric character. Although a symbol may be any length, only the first six characters are considered, and any additional characters are ignored. Consequently symbols which are identical in their first six characters are considered identical.

Pseudo-ops have fixed meanings, and cannot be redefined by the programmer.

The Assembler has in its permanent symbol table definitions of the symbols for all PDP-8 pseudo-op codes, memory reference, operate and IOT (Input/Output Transfer) instructions, which may be used without prior definition by the user. All other symbols must be defined in the source program. For example:

1. Permanent symbols:
   HLT is a symbolic instruction whose value of 7402 is taken by the Assembler from the permanent symbol table.
2. User defined symbols:
   A is a user defined symbol. When used as a symbolic address label, its value is the address of the instruction it precedes. This value is assigned by the Assembler. The user may assign values to symbols by using a direct assignment statement of the form A = 1234, which will be explained later.

INTERNAL SYMBOL REPRESENTATION FOR PAL III

Each permanent and user defined symbol occupies four words in the symbol table storage area, shown as follows:

| | | |
|---|---|---|
| Word 1 | C1 | C2 |
| Word 2 | C3 | C4 |
| Word 3 | C5 | C6 |
| Word 4 | | Octal code or address |

where C1, C2, . . . , C6 represent the first character, second character, . . . , sixth character respectively. For a permanent symbol, word 4 contains the octal code of the symbol, while for a user defined symbol, word 4 contains the address of the symbol. As an example, the permanent symbol TAD is represented as follows:

Word 1 = $24_8 \times 100_8 + 01 = 2401_8$ or TA
Word 2 = $04_8 \times 100_8 + 00 = 0400_8$ or D
Word 3 = 0000
Word 4 = 1000 (octal code for TAD)

The PAL III Assembler distinguishes between pseudo-ops, memory reference instructions, other permanent symbols, and user defined symbols by their relative position in the symbol table.

## SYMBOLIC ADDRESSES

A symbol used as a label to specify a symbolic address must appear as the first term in a statement and must be immediately followed by a comma. When used in this way, a symbol is assigned a value equal to the current location counter and is said to be defined. Permanent symbols (instructions, special characters, and pseudo-ops) may not be used as symbolic addresses.

A defined symbol can be used as an operand, or as a reference to an instruction. The user sets or resets the location counter by typing an asterisk followed by the octal absolute address value in which the next program word is to be stored. If the origin is not set by the user, PAL III begins assigning addresses at location 200.

```
          *300      /SET LOCATION COUNTER TO 300
TAG,      CLA
          JMP A
B,        0
A,        DCA B
          .
          .
          .
```

The symbol TAG (in the preceding example) is assigned a value of 0300, the symbol B a value of 0302, and the symbol A a value of 0303.

If a symbol is defined more than once in this manner, the Assembler will print the duplicate tag diagnostic:

DT xxxx AT nnnn

where xxxx is the symbol, and nnnn is the value of the location counter at the second occurrence of the symbol definition. The symbol is *not* redefined. For example:

```
        *300
START,  TAD A
        DCA COUNTER
CONTIN, JMS LEAVE
        JMP START
A,      -74
COUNTER,0
START,  CLA CLL
        .
        .
        .
```

The symbol START would have a value of 0300, the symbol CONTIN would have a value of 0302, the symbol A would have a value of 0304, the symbol COUNTER (considered COUNTE by the Assembler) would have a value of 0305. When the Assembler processed the next line it would print (during pass 1):

```
DT  START  AT  0306
```

Since the first pass of PAL III is used to define all symbols in the symbol table, the Assembler will print a diagnostic if, at the end of pass 1, there are any symbols remaining undefined. For example:

```
        *7170
A,      TAD C
        CLA CMA
        HLT
        JMP A1
C,      0
$
```

(The dollar sign must terminate all PDP-8 4K assembly programs.)

would produce the undefined address diagnostic:

UA xxxx AT nnnn

where xxxx is the symbol and nnnn is the location at which it was first seen. The user's entire symbol table is printed in alphabetical

order at the end of pass 1. In the case of the preceding example, this would look as follows:

```
A        7170
UA   A1  AT  7173
C        7174
```

The following are examples of legal symbolic addresses:

ADDR,
TOTAL,
SUM,
A1,

The following are examples of illegal symbolic addresses:

| | |
|---|---|
| AD)M, | (contains an illegal character) |
| 7ABC, | (first character must be alphabetic) |
| LA BEL, | (must *not* contain imbedded space) |
| D+TAG, | (contains a legal but non-alphanumeric character) |
| LABEL , | (must be terminated by a comma with no intervening spaces) |

## SYMBOLIC INSTRUCTIONS

Symbols used as instructions must be predefined by the Assembler or by the programmer. If a statement has no label, the instruction may appear first in the statement, and must be terminated by a space, tab, semicolon, slash, or carriage return. The following are examples of legal operators:

| | |
|---|---|
| TAD | (a mnemonic machine instruction operator) |
| PAGE | (an Assembler pseudo-op) |
| ZIP | (legal only if defined by the user) |

## SYMBOLIC OPERANDS

Symbols used as operands normally have a value defined by the user. The Assembler allows symbolic references to instructions or data defined elsewhere in the program. Operands may be numbers or expressions.

```
TOTAL,  TAD AC1+TAG
```

The values of the two symbols AC1 and TAG, already defined by the user, are combined by a two's complement add. This value is used as the address of the operand.

## SYMBOL TABLE

The Assembler processes user defined symbols in source program statements by adding them to its symbol table. The symbol table contains all defined symbols along with the binary value assigned to each symbol.

Initially, the Assembler's symbol table contains the mnemonic op-codes of the machine instructions and the Assembler pseudo-op codes, as listed in Appendix C. As the source program is processed, user defined symbols are added to the symbol table.

If, during pass 1, PAL III detects that the symbol table is full (in other words, there is no more memory space to store symbols and their associated values), the symbol table full diagnostic:

ST xxxx AT nnnn

is printed; xxxx is the symbol that caused the overflow condition and nnnn is the current location when the overflow occurred. The Assembler halts and may not be restarted.

More address arithmetic should be used to reduce the number of symbols. It is also possible to segment a program and assemble the segments separately, taking care to generate proper links between the segments. (See the MACRO-8 section Dealing with a Limited Symbol Space.) PAL III's symbol capacity when using the high-speed reader is 558 symbols. The permanent symbol table contains 80 symbols, leaving space for 478 possible user-defined symbols. When using the low-speed reader, PAL III's symbol capacity is 656 symbols, leaving space for 576 user-defined symbols.

## DIRECT ASSIGNMENT STATEMENTS

The programmer inserts new symbols with their assigned values directly into the symbol table by using a direct assignment statement of the form:

SYMBOL = VALUE

VALUE may be a number or expression. No space(s) or tab(s) may appear between the symbol to the left of the equal sign and the equal sign. The following are examples of direct assignment statements:

```
A=6
EXIT=JMP  I  Ø
C=A+B
```

All symbols to the right of the equal sign must be already defined. The symbol to the left of the equal sign is subject to the same restrictions as a symbolic address, and its associated value is stored in the user's symbol table. The use of the equal sign does not increment the location counter. It is, rather, an instruction to the Assembler itself.

A direct assignment statement may also equate a new symbol to the value assigned to a previously defined symbol. In this case the two symbols share the same memory location.

```
BETA=17
GAMMA=BETA
```

The new symbol, GAMMA, is entered into the user's symbol table with the value 17.

The value assigned to a symbol may be changed as follows:

```
ALPHA=5
ALPHA=7
```

The second line of code shown changes the value assigned to ALPHA from 5 to 7. (This will generate an RD error, explained below.)

Symbols defined by use of the equal sign may be used in any valid expression. For example:

```
*200
A=100    /DOES NOT UPDATE CLC
B=400    /DOES NOT UPDATE CLC
A+B      /THE VALUE 500 IS ASSEMBLED AT LOC. 200
TAD A    /THE VALUE 1100 IS ASSEMBLED AT LOC. 201
```

If the symbol to the left of the equal sign has already been defined, the redefinition diagnostic:

RD xxxx AT nnnn

will be printed as a warning, where xxxx is the symbol name and nnnn is the value of the location counter at the point of

redefinition. The new value will be stored in the symbol table; for example:

```
CLA=7600
```

will cause the diagnostic:

```
RD   CLA   AT   0200
```

Whenever CLA is used after this point, it will have the value 7600.

Multiple assignments can be carried to two levels only. Where X is some previously defined symbol or combination of symbols:

| | |
|---|---|
| A=B=X | will assemble, but |
| A=B=C=X | will not assemble. |

An error of this type will cause a "Pushdown Stack Overflow" diagnostic of the form:

PO xxxx AT nnnn

This is a non-recoverable error condition which causes the assembly to terminate. Continuation is not possible at this point. The error must be corrected and the assembly restarted.

The expression to the right of the rightmost equal sign must be composed completely of numbers and/or previously defined symbols.

The omission of the tag in a direct assignment statement will cause various errors, depending on the placement of the statement. If a statement such as:

=3

occurs before any symbol has been used in a program, PAL III will generate an RD error message during pass 1, with a meaningless printout for the symbol being defined. If the statement =3 occurs after a symbol has been used, the assembler assumes the last symbol referenced is being redefined. For instance:

```
*200
        CLA
        =3
```

will cause the diagnostic:

```
RD  CLA  AT   0201
```

during pass 1. In either case, PAL III may be restarted after pass 1, but attempting to continue to pass 2 will leave the Assembler in a state from which it can neither be continued for pass 3 nor re-started for another pass 1.

**Expressions**

Symbols and numbers are combined by arithmetic and logical operators to form expressions. There are three operators:

| | | |
|---|---|---|
| + | plus | Signifies two's complement addition |
| − | minus | Signifies two's complement subtraction |
| | space | Space is interpreted in context |

When a space occurs in an expression that does not contain a memory reference instruction, it means an inclusive OR is to be performed. For example:

```
CLA CLL
```

The symbol CLA has a value of 7200 and the symbol CLL has a value of 7100; CLA CLL would produce 7300. User defined symbols are treated as operate instructions. For example:

```
          A=333
          *222
B,        CLA
```

Possible expressions and their values using the symbols just defined are shown below. Notice that the Assembler reduces each expression to one 4-digit (octal) word:

| | | |
|---|---|---|
| A | 0333 | |
| B | 0222 | |
| A+B | 0555 | |
| A−B | 0111 | |
| −A | 7445 | |
| 1−B | 7557 | |
| B−1 | 0221 | |
| A B | 0333 | (an inclusive OR is performed) |
| −71 | 7707 | |
| etc. | | |

An expression is terminated by either a comma, carriage return, or semicolon. If the information generated is to be loaded, the current location counter is incremented. For example:

```
B-7; A+4; A+B
```

produces three words of information and the current location counter is incremented after each expression. The statement:

```
HALT=HLT CLA
```

produces no information to be loaded (it produces an association in the symbol table) and hence does not increment the current location counter.

```
        *4721
TEMP,
TEM2,   Ø
```

The location counter is not incremented after the line TEMP; the two symbols TEMP and TEM2 are assigned the same value, in this case 4721.

Since a PDP-8 instruction has an operation code of three bits as well as an indirect bit, a page bit, and seven address bits, the Assembler must combine memory reference instructions in a manner somewhat differently from the way in which it combines operate or IOT instructions. The Assembler differentiates between the symbols in its permanent symbol table and user defined symbols. The following symbols are used as memory reference instructions:

| | | |
|---|---|---|
| AND | 0000 | Logical AND |
| TAD | 1000 | Two's complement addition |
| ISZ | 2000 | Increment and skip if zero |
| DCA | 3000 | Deposit and clear accumulator |
| JMS | 4000 | Jump to subroutine |
| JMP | 5000 | Jump |
| FADD | 1000 | Floating addition |
| FSUB | 2000 | Floating subtraction |
| FMPY | 3000 | Floating multiply |
| FDIV | 4000 | Floating divide |
| FGET | 5000 | Floating GET |
| FPUT | 6000 | Floating PUT |

When the Assembler has processed one of these symbols, the space following it acts as an address field delimiter.

```
        *4100
        JMP A
A,      CLA
```

A has the value 4101, JMP has the value 5000, and the space acts as a field delimiter. These symbols are represented as follows:

| | |
|---|---|
| A | 100 001 000 001 |
| JMP | 101 000 000 000 |

The seven address bits of A are taken, i.e.:

000 001 000 001

The remaining bits of the address are tested to see if they are zeros (page zero reference); if they are not, the current page bit is set:

000 011 000 001

The operation code is then ORed into the JMP expression to form:

101 011 000 001

or, written more concisely in octal:

5301

In addition to the above tests, the page bits of the address field are compared with the page bits of the current location counter. If the page bits of the address field are nonzero and do not equal the page bits of the current location counter, an out-of-page reference is being attempted and the illegal reference diagnostic is printed on pass 2 or pass 3. For example:

```
        *4100
A,      CLA CLL
        .
        .
        .
        *7200
        JMP A
```

The symbol in the address field of the JMP instruction has a value of 4100 while the location counter (the address where the

instruction is placed in memory) has a value of 7200. This instruction is illegal because PAL III does not generate off-page references, and will be flagged during pass 2 or pass 3 by the illegal reference diagnostic:

```
IR  4100  AT  7200
```

**NOTE**

Such a diagnostic would not be generated when using MACRO-8, which automatically generates off-page references.

**Address Assignments**

The PAL III Assembler sets the origin, or starting address, of the source program to absolute location (address) 0200 unless the origin is otherwise specified by the programmer. As source statements are processed, PAL III assigns consecutive memory addresses to the instructions and data words of the object program. This is done by automatically incrementing the current location counter each time a memory location is assigned. A statement which generates a single object program storage word increments the location counter by one. Another statement might generate six storage words, incrementing the location counter by six.

Direct assignment statements and some Assembler pseudo-ops do not generate storage words and therefore do not affect the location counter.

CURRENT ADDRESS INDICATOR

The special character period (.) always has a value equal to the value of the current location counter. It may be used as any integer or symbol (except to the left of an equal sign), and must be preceded by a space when used as an operand. For example:

```
*200
JMP .+2
```

is equivalent to JMP 0202. Also,

```
*300
.+2400
```

will produce in location 0300 the quantity 2700. Consider:

```
*2200
CALL=JMS I .
0027
```

The second line (CALL=JMS I .) does not increment the current location counter, therefore, 0027 is placed in location 2200 and CALL is placed in the user's symbol table with an associated value of 4600 (the octal equivalent of JMS I .).

## INDIRECT ADDRESSING

When the character I appears in a statement between a memory reference instruction and an operand, the operand is interpreted as the address (or location) *containing* the address of the operand to be used in the current statement. Consider:

```
TAD 40
```

which is a direct address statement, where 40 is interpreted as the address on page zero containing the quantity to be added to the accumulator. References to addresses on the current page and to page zero may be done directly. An alternate way to note the page zero reference is with the letter Z, as follows:

```
TAD Z 40
```

This is an optional notation, not differing in effect from the previous example. Thus, if address 40 contains 0432, then 0432 is added to the accumulator. Now consider:

```
TAD I 40
```

which is an indirect address statement, where 40 is interpreted as the address of the address containing the quantity to be added to the accumulator. Thus, if address 40 contains 0432, and address 432 contains 0456, then 456 is added to the accumulator.

**NOTE**

Because the letter I is used to indicate indirect addressing, it is never used as a variable. Likewise the letter Z, which is sometimes used to indicate a page zero reference, is never used as a variable.

AUTOINDEXING

Interpage references are often necessary for obtaining operands when processing large amounts of data. The PDP-8 computers have facilities to ease the addressing of this data. When one of the absolute locations from 10 to 17 (octal) is indirectly addressed, the contents of the location is incremented before it is used as an address and the incremented number is left in the location. This allows the programmer to address consecutive memory locations using a minimum of statements.

It must be remembered that initially these locations (10 to 17 on page 0) must be set to one less than the first desired address. Because of their characteristics, these locations are called autoindex registers. No incrementation takes place when locations 10 to 17 are addressed directly. For example, if the instruction to be executed next is in location 300 and the data to be referenced is on the page starting at location 5000, autoindex register 10 can be used to address the data as follows:

```
0276    1377    TAD C4777     /=5000-1
0277    3010    DCA 10        /SET UP AUTO INDEX
0300    1410    TAD I 10      /INCREMENT TO 5000
.       .       .             /BEFORE USE AS AN ADDRESS
.       .       .
.       .       .
0377    4777    C4777,4777
```

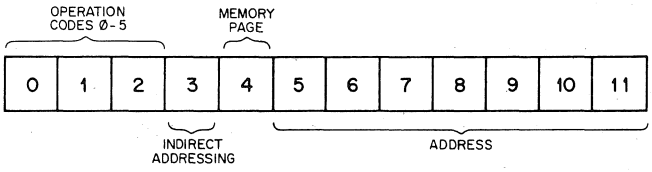
When the instruction in location 300 is executed, the contents of location 10 will be incremented to 5000 and the contents of location 5000 will be added to the contents of the accumulator. When the instruction TAD I 10 is executed again, the contents of location 5001 will be added to the accumulator, and so on.

**Instructions**

There are two basic groups of instructions: memory reference and microinstructions. Memory reference instructions require an operand; microinstructions do not require an operand.

## MEMORY REFERENCE INSTRUCTIONS

In PDP-8 computers, some instructions require a reference to memory. They are appropriately designated memory reference instructions, and take the following format:




Memory Reference Instruction Bit Assignments

Bits 0 through 2 contain the operation code of the instruction to be performed. Bit 3 tells the computer if the instruction is indirect, that is, if the address of the instruction specifies the location of the address of the operand. Bit 4 tells the computer if the instruction is referencing the current page or page zero. This leaves bits 5 through 11 (7 bits) to specify an address. In these 7 bits, 200 octal or 128 decimal locations can be specified; the page bit increases accessible locations to 400 octal or 256 decimal. For a list of the memory reference instructions and their codes, see Appendix C.

In PAL III a memory reference instruction must be followed by a space(s) or tab(s), an optional I or Z designation, and any valid expression, and may be defined with the FIXMRI instruction, explained under the section on Altering the Permanent Symbol Table. Permanent symbols may be defined using the FIXTAB instruction, and may be used in address fields as shown below:

```
A=1234
FIXTAB
TAD A
```

## MICROINSTRUCTIONS

Microinstructions are divided into two groups: operate and Input/Output Transfer (IOT) microinstructions.

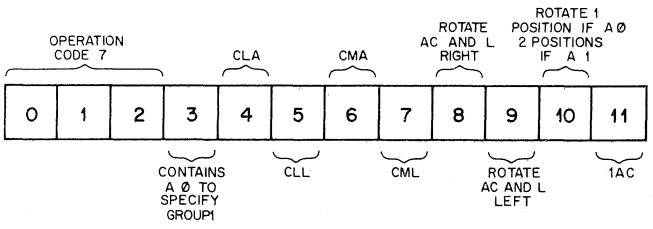
**NOTE**

If a programmer mistakenly makes an illegal combination of microinstructions, the Assembler will perform an inclusive OR between them; for example:

CLL SKP is interpreted as SPA
(7100 7410) (7510)

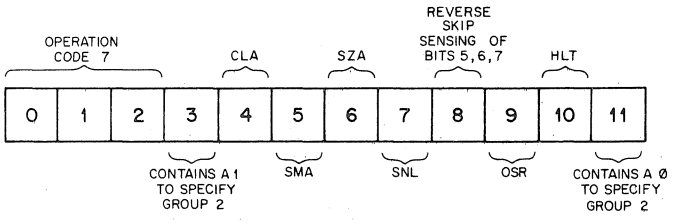
*Operate Microinstructions*

Within the operate group, there are two groups of microinstructions which cannot be mixed. Group 1 microinstructions perform clear, complement, rotate and increment operations, and are designated by the presence of a 0 in bit 3 of the machine instruction word. (See Permanent Symbol Table list in Appendix C.)




Group 1 Operate Microinstruction Bit Assignments

Group 2 microinstructions check the contents of the accumulator and link and, based on the check, continue to or skip the next instruction. Group 2 microinstructions are identified by the presence of a 1 in bit 3 and a 0 in bit 11 of the machine instruction word (See Appendix C).




Group 2 Operate Microinstruction Bit Assignments

Group 1 and Group 2 microinstructions cannot be combined because bit 3 determines either one or the other.

Within Group 2, there are two groups of skip instructions. They can be referred to as the OR group and the AND group.

| OR Group | AND Group |
|---|---|
| SMA | SPA |
| SZA | SNA |
| SNL | SZL |

The OR group is designated by a 0 in bit 8, and the AND group by a 1 in bit 8. OR and AND group instructions cannot be combined because bit 8 determines either one or the other.

If the programmer does combine legal skip instructions, it is important to note the conditions under which a skip may occur.

1. OR Group—If these skips are combined in a statement, the inclusive OR of the conditions determines the skip. For example:

```
SZA SNL
```

   The next statement is skipped if the accumulator contains 0000, or the link is a 1, or both conditions exist.

2. AND Group—If the skips are combined in a statement, the logical AND of the conditions determines the skip. For example:

```
SNA SZL
```

   The next statement is skipped only if the accumulator differs from 0000 and the link is 0.

*Input/Output Transfer Microinstructions*

These microinstructions initiate operation of peripheral equipment and effect an information transfer between the central processor and the Input/Output device(s).

The Permanent Symbol Table in Appendix C contains the commonly used in IOTs for the disk, TTY, and high speed devices. These and other IOTs are discussed in detail in the *Small Computer Handbook*.

## PSEUDO-OPERATORS

The programmer uses pseudo-operators to direct the Assembler to perform certain tasks or to interpret subsequent coding in a certain manner. Some pseudo-ops generate storage words in the object program, other pseudo-ops direct the Assembler as to how to proceed with the assembly. Pseudo-ops are maintained in the permanent symbol table; pseudo-ops should not be used as variable names within a program.

The function of each PAL III pseudo-op is described below.

*Indirect Addressing*

I Symbolic representation for indirect addressing, must be separated on each side by at least one space.

For example:

```
DCA I ADD
```

The value of the symbol ADD is used as the address of the address in which the contents of the accumulator will be stored.

Z Optional method of denoting a page zero reference.

For example:

```
DCA ADD
DCA Z ADD
```

The two statements above have the same meaning and generate the same code, where ADD is on page zero.

Both Z and I can be present in the same instruction, separated by at least one space, as follows:

```
DCA Z I ADD
```

which is the same as:

```
DCA I ADD
```

*Radix Control*

Numbers used in a source program are initially considered to be

octal numbers. However, if the programmer wishes to have certain numbers interpreted as decimal, he can use the pseudo-op DECIMAL.

| | |
|---|---|
| DECIMAL | All following numbers are taken as decimal until the occurrence of the pseudo-op OCTAL. |
| OCTAL | Resets the radix to its original octal base. |

*Extended Memory*

When using more than one memory bank, the pseudo-op FIELD instructs the Assembler to output a field setting. This field setting is punched during pass 2 and is recognized by the Binary Loader, which in turn causes all subsequent information to be loaded into the field specified by the expression.

| | |
|---|---|
| FIELD n | Where n is an integer, a previously defined symbol, or an expression within the range $0 \leqslant n \leqslant 7$. |

The FIELD pseudo-op causes a field setting (binary word) of the form:

$$11\ xxx\ 000 \quad \text{where} \quad 000 \leqslant xxx \leqslant 111_2$$

to be output on the binary tape during pass 2 followed by an origin setting of 200. This word is read by the Loader, which then begins loading information into the new field.

The field setting is never remembered by the Assembler, and no initial field setting is punched. A binary tape produced without field settings may be loaded into any one field of core by appropriate manipulation of the Data Field switches when using the Binary Loader. A symbol in one field may be used to reference the same location in any other field. The field to which it refers is determined by the use of the CDF and CIF instructions. (The programmer who is unfamiliar with the IOTs but wishes to use them should experiment with several short test programs to satisfy himself as to their effect.) An example of this method of symbol space conservation might be:

```
                    *200
0200    0005        DATA,   5
0201    0006                6
0202    0007                7
0203    0004                4
                    *300
0300    1200                TAD DATA     /YIELDS 5 IN AC
0301    1203                TAD DATA +3  /YIELDS 10 IN AC
                            CDF 10
                            CIF 10
                    FIELD 1
0200    0006                6
0201    0007                7
0202    0003                3
0203    0002                2
0204    1200                TAD DATA     /YIELDS 6 IN AC
0205    7200                CLA
0206    1204                TAD DATA+4   /YIELDS 1200 IN AC
```

**NOTE**

CDF and CIF instructions must be used prior to any instruction referencing a location outside of the current field, as shown in the following example:

```
            *200
            TAD P301
            CDF 00
            CIF 10
            JMS PRINT
            CIF 10
            JMP NEXT
P301,       301
            FIELD 1
            *200
NEXT,       TAD P302
            CDF 10
            JMS PRINT
            HLT
P302,       302
PRINT,      0
            TLS
            TSF
            JMP .-1
            CLA
            RDF
            TAD P6203
            DCA .+1
            000
            JMP I PRINT
P6203,      6203
```

When FIELD is used, the Assembler follows the new FIELD setting with an origin at location 200. For this reason, if the programmer wants to assemble code at location 400 in field 1 he must write:

```
FIELD 1          /CORRECT EXAMPLE
*400
```

The following is *incorrect* and will not generate the desired code:

```
*400             /INCORRECT
FIELD 1
```

*End of Tape*

The pseudo-op PAUSE signals the Assembler to stop processing the paper tape being read. The current pass is not terminated, and processing continues when the user depresses the CONTinue key.

When processing a segmented program, the programmer uses the PAUSE pseudo-op as the last statement of each segment (tape) to halt Assembler processing, giving him time to insert the next segment of his program.

The PAUSE pseudo-op should be used *only* at the physical end of a tape or file and with two or more tapes of one program. When a PAUSE statement is reached,

1. The Assembler stops.
2. This is the physical end of the tape; the Assembler resets the input buffer pointer.
3. Operator intervention is required to put the next tape segment of the program in the reader and press the CONTinue key.

If a PAUSE is encountered somewhere other than at the physical end of a tape, some of the user code immediately after the PAUSE will not be assembled. This occurs because PAL III has an input buffer to allow maximum use of reader speed. A tape is read in until the buffer is filled or the physical end of the tape is reached.

The contents of the buffer are then processed. However, upon recognizing a PAUSE, PAL III resets the buffer to empty and waits for step 3 above.

*End of Program*

The special symbol dollar sign ($) indicates the end of a program. When the Assembler encounters the dollar sign, it terminates the current pass. The Assembler must read a $ after each pass before it will correctly proceed with the assembly.

*Altering the Permanent Symbol Table*

PAL III contains a table of symbol definitions for the PDP-8 and its most common peripheral devices. These are symbols such as TAD, DCA, and CLA, which are used in most PDP-8 programs. This table is considered to be the permanent symbol table for PAL III; all of the symbols it contains are listed in Appendix C.

If the user purchases one or more optional devices whose instruction set is not defined among the permanent symbols (for example EAE or an A/D Converter), he would want to add the necessary symbol definitions to the permanent symbol table in every program he assembles. Conversely, the user who needs more space for user defined symbols would probably want to delete all definitions except the ones used in his program. For such purposes, PAL III has three pseudo-ops that can be used to alter the permanent symbol table. These pseudo-ops are recognized by the Assembler only during pass 1. During either pass 2 or pass 3 they are ignored and have no effect.

| | |
|---|---|
| EXPUNGE | Deletes the entire permanent symbol table, except pseudo-ops. |
| FIXTAB | Appends all presently defined symbols to the permanent symbol table. All symbols defined before the occurrence of FIXTAB are made part of the permanent symbol table until the Assembler is reloaded. For example, the PAL III Extended Symbols Tape ends with FIXTAB. |

To append the following RF08 disk IOTs to the symbol table, the programmer generates an ASCII tape of:

```
DCIM=6611
DIML=6615
DIMA=6616
DFSE=6621
DISK=6623
DCXA=6641
DXAL=6643
DXAC=6645
DMMT=6646
FIXTAB
PAUSE
```

The ASCII tape is then read into core ahead of the symbolic program tape during pass 1. The PAUSE pseudo-op stops assembly, and the Loader waits for the programmer to put the symbolic program tape into the tape reader and press CONTinue.

Each time the Assembler is loaded, PAL III's permanent symbol table is restored to contain only the permanent symbols shown in Appendix C.

After altering the symbol table to fit his needs, the user might want to keep PAL III in this state for future use. This can be done by punching a binary of the section of core occupied by PAL III with its new symbol table.

To do this:

1. Read in PAL III and modify symbol table as desired.
2. PAL III's symbol table begins at location 2332 (octal). Count all the symbols in the altered symbol table. Since each symbol and its value require four words, multiply this number by 4. Convert this number to octal and add it to 2332 (octal). This number is the upper limit of PAL III. The lower limit is 0001.
3. Using the Binary Punch Routine (DEC-08-YX1A-PB), which does a binary core dump to the high-speed or Teletype punch, and the limits as stated in 2 above, punch out the PAL III Assembler.
4. The output of the Binary Punch Routine is the Assembler with the modified symbol table and can be loaded with the Binary Loader. This revised version of the Assembler can thereafter be used instead of the original version.

The third pseudo-op used to alter the permanent symbol table in PAL III (and not present in MACRO-8) is FIXMRI which may be used only after an EXPUNGE instruction:

FIXMRI Fix memory reference instruction. Memory reference instructions are stored in the permanent symbol table immediately following the pseudo-ops. The letters FIXMRI must be followed by one space, the symbol for the instruction to be defined, an equal sign, and the value of the symbol. The pseudo-op must be repeated for each memory reference instruction to be defined. All memory reference instructions must be defined before the definition of any other symbols.

For example:

```
EXPUNGE
FIXMRI TAD=1000
FIXMRI DCA=3000
CLA=7200
FIXTAB
PAUSE
```

When the preceding program segment is read into the Assembler during pass 1, all symbol definitions are deleted and the three symbols listed are added to the permanent symbol table. Notice that CLA is not a memory reference instruction. This process is often performed to alter the Assembler's symbol table so that it contains only those symbols used at a given installation or by a given program. This may increase the Assembler's capacity for user defined symbols in the program.

**Program Preparation and Assembler Output**

In PAL III and MACRO-8, the source language or symbolic tape is punched in ASCII code on 8-channel paper tape, using an off-line Model LT-33 Teletype or the on-line Symbolic Editor. In general, a program should begin with leader code, which may be blank tape, code 200, or RUBOUTs.

Certain codes which the Assembler ignores may be used freely to produce a more readable symbolic program listing. These codes are TAB and LINE FEED. The Assembler also ignores extraneous spaces, carriage return/line feed combinations, and blank tape. When the Assembler encounters a form feed character, it causes 12 blank lines to be output on the listing (in PAL III only).

The two programs below are identical and produce the same binary code. The second, however, was generated using the TAB

function of the Symbolic Editor and is easier to read. The first program assembles faster only because there is less paper tape to be read into the computer.

Program #1:

```
*200
/EXAMPLE OF INPUT TO THE FORMAT
/GENERATOR PROGRAM
BEGIN, 0/START OF PROGRAM
KCC
KSF/WAIT FOR FLAG
JMP .-1/FLAG NOT SET YET
KRB/READ IN CHARACTER
DCA CHAR
TAD CHAR
TAD MSPACE/IS IT A SPACE?
SNA CLA
HLT/YES
JMP BEGIN+2/NO:INPUT AGAIN
CHAR,0/TEMPORARY STORAGE
MSPACE,-240/-ASCII EQUIVALENT
/END OF EXAMPLE
$
```

Program #2:

```
*200
/EXAMPLE OF INPUT TO THE FORMAT
/GENERATOR PROGRAM
BEGIN,  0               /START OF PROGRAM
        KCC
        KSF             /WAIT FOR FLAG
        JMP .-1         /FLAG NOT SET YET
        KRB             /READ IN CHARACTER
        DCA CHAR
        TAD CHAR
        TAD MSPACE      /IS IT A SPACE?
        SNA CLA
        HLT             /YES
        JMP BEGIN+2     /NO: INPUT AGAIN
CHAR,   0               /TEMPORARY STORAGE
MSPACE, -240            /-ASCII EQUIVALENT
/END OF EXAMPLE
$
```

The program consists of statements and pseudo-ops, is terminated by the dollar sign ($), and followed by some trailer code. If the program is large, it can be segmented using the pseudo-op PAUSE, which often facilitates the editing of the source program since each section will be physically smaller.

The Assembler initially sets the current location counter to 0200. This counter is reset whenever the asterisk (*) is processed.

During pass 1, all illegal characters cause a diagnostic to be printed. The tape should be corrected and reassembled.

The Assembler reads the source tape and defines all symbols used. The user's symbol table is printed (or punched) at the end of pass 1. The symbol table is printed in alphabetical order. If any symbols remain undefined, the undefined address diagnostic is printed. If the program listed on the previous page were assembled, the pass 1 symbol table output would be:

```
BEGIN   0200
CHAR    0213
MSPACE  0214
```

During pass 2, the Assembler reads the source tape and generates the binary code, using the symbol table equivalences defined during pass 1. The binary tape that is punched may be loaded by the Binary Loader. This binary tape consists of leader code, an origin setting, and data words. At the end of pass 2, a checksum is punched on the binary tape, and trailer code is generated. During pass 2, the Assembler may diagnose an illegal reference; when using the LT-33 punch, the diagnostic is both printed and punched, and is preceded and followed by RUBOUTs. The Binary Loader ignores everything that has been punched on a tape between RUBOUTs.

During pass 3, the Assembler reads the source tape and generates the code from the source statements. The assembly listing is printed (or punched). It consists of the current location counter, the generated code in octal, and the source statement. The symbol table is printed at the end of the pass. If the sample program listed above were assembled, the pass 3 output would be:

```
                        *200
                        /EXAMPLE OF INPUT TO THE FORMAT
                        /GENERATOR PROGRAM
0200    0000  BEGIN,     0                  /START OF PROGRAM
0201    6032              KCC
0202    6031              KSF                /WAIT FOR FLAG
0203    5202              JMP .-1            /FLAG NOT SET YET
0204    6036              KRB                /READ IN CHARACTER
0205    3213              DCA CHAR
0206    1213              TAD CHAR
0207    1214              TAD MSPACE         /IS IT A SPACE?
0210    7650              SNA CLA
0211    7402              HLT                /YES
0212    5202              JMP BEGIN+2        /NO: INPUT AGAIN
0213    0000  CHAR,       0                  /TEMPORARY STORAGE
0214    7540  MSPACE,     -240               /ASCII EQUIVALENT
              /END OF EXAMPLE

BEGIN   0200
CHAR    0213
MSPACE  0214
```

**Operating Procedures**

The PAL III Assembler is provided to DEC customers as a binary tape, which is loaded into the PDP-8 memory by means of the Binary Loader, using either the LT-33 reader or the high-speed reader. The Assembler also uses either the LT-33 reader or the high-speed reader to read the source language tape, and it uses either the LT-33 punch or the high-speed punch for output. The selection of I/O devices is made when the Assembler is started. The source language tape must be in the proper reader, with the reader and punch turned on.

When using the high-speed punch, the symbol table is printed on the LT-33 Teletype if bit 11 of the switch register is a 0. The symbol table is punched on the high-speed punch if bit 11 of the switch register is a 1.

All diagnostics are printed on the LT-33 except for the undefined address diagnostic when using the LT-33 punch, or the high-speed punch if it is included in the machine configuration and turned on. The only diagnostic in pass 2 will be illegal reference. (Since this diagnostic is printed on the LT-33, it will also be punched on the binary tape. It will, however, be ignored by the Binary Loader.) The bit 11 switch option can also be used during

pass 3. If the machine is not equipped with a high-speed punch, bit 11 must be set to 0.

In addition to the binary tape of the PAL III Assembler, the user is provided with an ASCII tape (PAL III Extended Symbols Tape) containing symbol definitions for the instruction sets of the available options to the PDP-8 (card readers, magnetic tapes, and A/D converters). A limited amount of space is available in a 4K system; therefore, expanding the number of permanent symbols that the Assembler recognizes will decrease the maximum number of symbols the user has available.

The following is a description of steps in using the PAL III Assembler:

1. Load the Assembler, using either the LT-33 reader or the high-speed reader (see Appendix A).
2. Set 0200 into the Switch Register; press ADDRess LOAD.
3. Place the source language tape in the reader, turn on the appropriate reader and the punch.
4. Set bits 0 and 1 of the Switch Register for the proper pass. These settings are:

| Bit 0 | Bit 1 | |
|---|---|---|
| 0 | 1 | pass 1 |
| 1 | 0 | pass 2 |
| 1 | 1 | pass 3 |

Pass 1 is required so that the Assembler can initialize its symbol table and define all user symbols. After pass 1 has been made, either pass 2 or pass 3 can be made.

5. Bit 11 switch options:

| | | |
|---|---|---|
| pass 1 | Bit 11=1 | Punch the symbol table on the high-speed punch if it is in the machine configuration. |
| | Bit 11=0 | Print (and punch) the symbol table on the LT-33 (low-speed punch). |
| pass 2 | Bit 11=1 | Punch binary tape on high-speed punch. |
| | Bit 11=0 | Punch binary tape on low-speed punch. |

pass 3 Bit 11=1 Punch the assembly listing tape in ASCII, on the high-speed punch.

Bit 11=0 Print the assembly listing on the LT-33

Bit 10 switch options:

pass 3 Bit 10=1 Output TAB (code 211) as 8 space tab stops.

Bit 10=0 Output TAB as TAB RUBOUT (code 211 and 377).

Bit 2 switch options:

passes 1 and 3

Bit 2=1 Suppress output of symbol table.

Bit 2=0 Output symbol table.

6. Press CLEAR and CONTinue to begin pass 1 only. Press CONTinue to begin passes 2 or 3. The Assembler halts at the end of each pass. Proceed from step 3. If the Assembler has halted because of a PAUSE statement, put the next tape into the reader and press CONTinue.

## SUMMARY OF DIAGNOSTIC MESSAGES FOR PAL III

*Pass 1 Diagnostics*

The Assembler reads the source tape, defines all user symbols, and outputs the user symbol table in alphabetical order. Pass 1 diagnostics are:

IC xxxx AT nnnn Illegal Character

Where xxxx is the value of the illegal character and nnnn is the value of the current location counter when the character was processed.

RD xxxx AT nnnn Redefinition

Where xxxx is the symbol being redefined and nnnn is the value of the current location counter at the point of redefinition. The symbol is redefined.

DT xxxx AT nnnn Duplicate Tag

An attempt is being made to redefine a symbol using the comma. xxxx is the symbol and nnnn is the value of the current location counter at the point of redefinition. The previous value of the symbol is retained and the symbol is not redefined.

ST xxxx AT nnnn Symbol Table Full

Where xxxx is the symbol causing the overflow and nnnn is the value of the current location counter at the point of overflow. The Assembler halts and cannot be restarted.

PO xxxx AT nnnn Pushdown List Overflow

An attempt is being made to carry a multiple assignment to more than two levels; i.e., A=B=C=X. xxxx is the value of the pushdown stack pointer (an internal address in PAL III). The value of the current location counter when overflow is detected is nnnn. The Assembler halts at this point without reading more source tape. The CONTinue key has no effect at this time; however, the Assembler can be restarted at location 200, as indicated in step 2 under Operating Procedures.

UA xxxx AT nnnn Undefined Address

Where xxxx is the symbol that was used, but never defined, and nnnn is the value of the current location counter when the symbol was first processed. This message is printed with the symbol table at the end of pass 1. The symbol is assigned a value equal to the highest address on the memory page where it was first used.

*Pass 2 Diagnostics*

The Assembler reads the source tape, and, using the symbol table defined during pass 1, generates and punches the binary code. This binary tape can then be loaded by the Binary Loader. The pass 2 diagnostic is:

IR xxxx AT nnnn Illegal Reference

Where xxxx is the address being referenced and nnnn is the value of the current location counter. The illegal address is then treated as if it were on the proper memory page. For example:

```
*7306
JMP 307
```

would produce:

```
IR  0307  AT  7306
```

and would generate 5307 to be loaded into location 7306.

*Pass 3 Diagnostics*

The Assembler reads the source tape and, using the symbol table defined during pass 1, generates and prints the code represented by the source statements. The current location counter, the contents, and the source statement are printed side by side on one line. If bit 11 of the Switch Register is a 1 and the machine configuration includes the high-speed punch, the assembly listing is punched in ASCII. The pass 3 diagnostic is Illegal Reference, as in pass 2.

## MACRO-8 PROGRAMMING

Macro-8 is a 4K two-pass paper tape assembler similar to PAL III, which contains several additional features which may be of use to more advanced PDP-8 programmers. These features include link generation, literals, Boolean operations, double precision integer input, floating point input, a text input facility, and user-defined macros. The assembler is compatible in most respects with PAL III; the areas of difference are noted later in this manual.

### Characters

In addition to those characters discussed under PAL III, the following characters are used in MACRO-8.

| Symbol | Name | Function |
|---|---|---|
| & | Ampersand | Combines symbols or numbers (Boolean AND) |
| ! | Exclamation Point | Combines symbols or numbers (Boolean OR) |
| " | Double Quote | Generates 8-bit ASCII code |
| ( ) | Parentheses | Defines a literal on the current page |
| [ ] | Square Brackets | Defines a page 0 literal |
| < > | Angle Brackets | Defines a macro |

NOTE: On an LT-33 :

| | |
|---|---|
| [ | is generated by SHIFT/K |
| ] | is generated by SHIFT/M |
| < | is generated by SHIFT/, (comma) |
| > | is generated by SHIFT/. (period) |
| Line Feed | (code 212) is ignored, as in PAL III, unless it immediately precedes a dollar sign. The sequence line feed/dollar sign is taken as unconditional end-of-pass. |

### Expressions

All symbols and numbers (exclusive of pseudo-ops, macro names, and double precision or floating point constants), may be combined with certain arithmetic and logical operators to form expressions. These operators are:

| Operator | Name | Function |
|---|---|---|
| + | Plus | Signifies two's complement addition* |
| – | Minus | Signifies two's complement subtraction* |
| ! | Exclamation Point | Signifies Boolean inclusive OR (union)* |
| & | Ampersand | Signifies Boolean AND (intersection)* |
| | Space | Interpreted in context; can signify an inclusive OR, or act as a field delimiter as in PAL III |

*As explained in *Introduction to Programming.*

**NOTE**

Expressions may not contain pseudo-ops other than I and Z, macro names, double precision integers, or floating point constants. To do so is an error, and erroneous code may result without an error message being given.

Symbols and integers may be combined with any of the preceding operators. A symbolic expression is evaluated from left to right with one exception—space, which serves as a delimiter. Grouping of terms via the use of parentheses is not permitted in MACRO-8. Consider the following examples:

| | A | B | A+B | A–B | A!B | A&B |
|---|---|---|---|---|---|---|
| VALUE | 0002 | 0003 | 0005 | 7777 | 0003 | 0002 |
| VALUE | 0007 | 0005 | 0014 | 0002 | 0007 | 0005 |
| VALUE | 0700 | 0007 | 0707 | 0671 | 0707 | 0000 |

Since space is treated as a delimiter, separating two terms of an expression, the two expressions:

```
A!B+C
A B+C
```

are not equivalent. In the first case the assembler interprets the expression from left to right as follows:

```
(A!B)+C
```

In the second case, each term is considered separately and the results combined with an inclusive OR:

```
A (B+C)
```

As in the following example, the values of the two expressions will differ:

$$3!4+5 = 7+5 = 14_8$$
$$3\ 4+5 = 3!11_8 = 13_8$$

**NOTE**

Expressions of the form A +B and A+ B are incorrect and may cause erroneous code to be generated. Symbols and integers must be combined with a single operator separating them.

**Origin Setting**

The origin is ordinarily set by use of the special character asterisk (*) as described in PAL III. All symbols to the right of the asterisk must already have been defined. For example, if D has the value 250 then:

```
*D+1Ø
```

will set the location counter to 0260.

To ease the programmer's addressing problems, a convention has been defined that divides memory into sections called pages. Each page contains $200_8$ locations ($128_{10}$) numbered 0 to $177_8$ on that page. There are $40_8$ or $32_{10}$ pages numbered 0 to $37_8$. Some examples of page numbers and the absolute and relative locations (addresses) are shown below. See Chapter 2 of *Introduction to Programming* for a complete list of page numbers and their addresses. It must be remembered, however, that there is no physical separation of pages in memory.

| Page | Absolute Address | Relative Address |
|---|---|---|
| 0 | 0—177 | 0—177 |
| 1 | 200—377 | 0—177 |
| 2 | 400—577 | 0—177 |
| . | . | . |
| . | . | . |
| . | . | . |
| 36 | 7400—7777 | 0—177 |
| 37 | 7600—7777 | 0—177 |

To simplify page handling, the pseudo-op PAGE can be used:

PAGE n — The PAGE pseudo-op resets the location counter to the first address of page n, where n is an integer, a previously defined symbol, or a symbol expression, all of whose terms have been defined previously and whose value is from 0 to $37_8$ inclusive.
For example:

PAGE 2 Sets the location counter to 0400
PAGE 6 Sets the location counter to 1400

PAGE — When used without an argument, PAGE resets the location counter to the first location on the next succeeding page. Thus, if a program is being assembled into page 1 and the programmer wishes to begin the next segment on page 2 he need only insert the pseudo-op PAGE, as follows:

```
*200
        .
        .
        JMP .-7
PAGE
        CLA
        .
        .
```

In this case, the CLA will be assembled into location 400.

If, when the PAGE pseudo-op is used without an argument, the current location counter is at the first location of a page, it will not

be moved. In the following example, the code TAD B is assembled into location 400:

```
*377
        JMP .-3
PAGE
        TAD B
```

If several consecutive PAGE pseudo-ops are given, the first will cause the current location counter to be reset as specified. The rest of the PAGE pseudo-ops in the sequence will be ignored.

**NOTE**

Since PAGE is a pseudo-op in MACRO-8, it is illegal as a tag. Therefore the coding:

```
PAGE,   Ø
        TAD PAGE
```

may not be used.

**Link Generation**

In addition to handling the symbolic addressing on the current page of core memory, MACRO-8 automatically generates links for off-page references. MACRO-8 compares the page bits of the address field with the page bits of the location counter. If the page bits of the address field are nonzero (not a page 0 reference) and do not equal the page bits of the location counter, an off-page reference is being attempted.

If reference is made to an address not on the page where the instruction is located, the assembler sets the indirect bit (bit 3), and an indirect address linkage will be generated on the current memory page. If the off-page reference is already an indirect one, the error diagnostic II (Illegal Indirect) will be generated during passes 2 and 3.

When a link is generated, the LG (Link Generated) message will be printed on passes 2 and 3. In the case of several off-page references to the same address, the link will be generated only once, but the LG message will be printed each time.

```
*2117
A,      CLA
        .
        .
*2600
        JMP  A
```

In the example above, the space preceding the user defined symbol A acts as an address field delimiter. The Assembler will recognize that the register labelled A is not on the current page (in this case 2600 to 2777) and will generate a link to it as follows:

1. In location 2600 the Assembler will place the word 5777 which is equivalent to JMP I 2777.
2. In address 2777 (the last available location on the current page) the Assembler will place the word 2117 (the actual address of A).

Although the Assembler will recognize and generate an indirect address linkage when necessary, the programmer may indicate an explicit indirect address by the pseudo-op I. This must be between the instruction code and the address field, as it would be placed in PAL III. The Assembler cannot generate a link for an instruction that is already specified as being an indirect reference and will print message II (Illegal Indirect). For example:

```
*2117
A,      CLA
        .
        .
*2600
        JMP  I A
```

The above coding will not work because A is not defined on the page where JMP I A is attempted and the indirect bit is already set.

**Literals**

Symbolic expressions appearing in the operand part of an instruction usually refer to locations containing the quantities being operated upon. Therefore, the programmer must explicitly reserve locations to hold his constants. The MACRO-8 language provides a means (known as literals) for using a constant directly. Suppose,

for example, that the programmer has an index which is to be incremented by two. One way of coding this operation would be as follows:

```
*200
        .
        .
        CLA
        TAD  INDEX
        TAD  C2
        DCA  INDEX
        .
        .
C2,     2
```

Using a literal, the same coding would be rewritten as:

```
*200
        .
        .
        CLA
        TAD  INDEX
        TAD  (2)
        DCA  INDEX
        .
        .
```

The left parenthesis is a signal to the Assembler that the expression following is to be evaluated and assigned a word in the constants table of the current page. This is the same table in which the indirect address linkages are stored. In the previous example, the quantity 2 is stored in a word in the linkage and literals list beginning at the top of the current memory page. The instruction in which the literal appears is encoded with an address referring to the address of the literal. A literal is assigned to storage the first time it is encountered; subsequent reference to that literal from the current page is made to the same register.

The use of literals in preference to the first method of handling constants frees symbol storage space for variables.

If the programmer wishes to assign literals to page zero rather than to the current page, he may use square brackets, [ and ], in place of the parentheses. This enables him to reference a single literal from any page of core. For example:

```
*200
        TAD [2]
        .
        .
*500
        TAD [2]
        .
        .
```

For the first and succeeding times the literal 2 is referenced, identical code is generated to a single location on page zero containing the literal. In this case, the following code would be generated:

| Location | Contents |
|---|---|
| 0177 | 0002 |
| 0200 | 1177 |
| 0500 | 1177 |

Whether on page zero or the current page, the right (closing) member is ignored and may be omitted. The following examples are acceptable:

```
TAD (777
AND [JMP
```

In the second example, the instruction AND [JMP has the same effect as:

```
AND [5000
```

Literals can be nested. For example:

```
*200
        TAD (TAD (30
```

will generate the following:

| Location | Contents |
|---|---|
| 0200 | 1376 |
| : | : |
| 0376 | 1377 |
| 0377 | 0030 |

This type of nesting can be carried to two levels as in the preceding example. Further nesting will cause BE (overlapping of internal tables) and US (undefined symbol) error messages to be generated.

Literals are stored on each page starting at page address 177 (relative) and extending toward page address 0 (relative). If a literal is generated for a nonzero page and the origin is then set to another page, the current page literal buffer is output (punched on pass 2; printed on pass 3). This does not affect later execution. The user may find that the literal buffer has been dumped before the end of a page to make room for more literals. The same literal will be generated if used after that point. If the origin is then reset to the previously used page, the same literal will be generated if used again, but it will not destroy previously used literals on that page. Literals and links are stored only as far down as the highest instruction on the page. Further attempts to define literals will result in a PE (Page Exceeded) or ZE (Page Zero Exceeded) error message. Links are not stored in empty locations preceding the highest location used for instruction.

To summarize, literals may take the following forms:

```
[C        (C
[V        (V
["A       ("A
[I        (I
[E        (E
```

where C is a constant, V is a variable, A is any single ASCII character other than blank tape (code 000) or leader-trailer (code 200), I is an instruction, and E is an arithmetic expression, [ indicates a page 0 reference and ( indicates a current page reference.

Arithmetic expressions may consist of constants, variables, and operators but must not include literals.

Single text characters may be combined with each other in an arithmetic expression but not with other constants or variables. For example:

```
("A+"B-"C
```

is equivalent to

```
(300
```

An instruction may contain a literal; for example:

```
TAD (JMP I [500
```

is valid; however:

```
TAD (A+(50
```

will not assemble since (A+ (50 is not a valid expression. Literals may be used as the address part of a memory reference instruction:

```
TAD (50
```

or in place of an instruction:

```
(MSG 4
```

which causes the location address of the literal (MSG4 to be assembled at the point where it occurs in the program.

**NOTE**

If a large number of nested literals or particularly large numbers of literals are used, the literal list may be output before the logical end of the page. This will not affect later execution.

**Field Pseudo-op**

In addition to punching the field setting and new origin setting (*200) as in PAL III, the FIELD pseudo-op in MACRO-8 first causes all current page literals and links, then all page 0 literals and links to be output. For this reason the following programming techniques should be considered:

1. Complete all coding in one field before moving to another. If you return to a previous field, the literals and links will not have been remembered.
2. Be exceedingly careful about referencing in one field a variable which was defined in another. The field of a variable is not stored (the assembler does not know what field a variable is in). For example:

```
                    FIELD 1
                    *300
10300    0340       A,          340
10301    1340                   TAD A
                    FIELD 2
20200    3340                   DCA A
```

In this example, the coding DCA A in field 2 caused octal 3340 to be generated, but this may be taken as a reference to location 340 in field 2.

The argument to the FIELD pseudo-op should be a value $0 \leqslant N \leqslant 7$. If N is greater than 7, the bits to the left of the low order digit will be ignored. Thus field 8 will be assembled as field 0 and flagged as an illegal character if the radix is octal.

On pass 3, the octal address will be preceded by a single digit denoting the current field. The first digit is merely a convenience for the programmer and has no other effect on the assembly.

**Text Facility**

SINGLE CHARACTER TEXT FACILITY

If a single character is preceded by a double quote ("), the 8-bit value of ASCII code for the character is inserted instead of interpreting the letter as a symbol. For example:

```
CLA
TAD ("A
```

will place the constant 0301 in the accumulator.

The code" . will be assembled as 0256.

If a carriage return is desired as the text character, it must not be the carriage return intended to end the line of code. To generate a 0215 with this facility, it is necessary to generate:

"[CARRIAGE RETURN/LINE FEED]
[CARRIAGE RETURN/LINE FEED]

or

"[CARRIAGE RETURN/LINE FEED]

or

"[CARRIAGE RETURN]
[CARRIAGE RETURN/LINE FEED]

The latter cannot be generated with the Symbolic Editor, which outputs a line feed after each carriage return. The Editor also cannot generate the sequence:

"[LINE FEED]
[CARRIAGE RETURN/LINE FEED]

which is necessary to generate the code 0212 with this facility. In this case it is best to use the octal 212 instead of "[LINE FEED].

On pass 3, certain characters, while causing the correct octal coding to be printed, will be interpreted as format control characters in the source listing. A summary of special cases follows:

| Source Code | Listing Appearance | | |
|---|---|---|---|
| *200 | | | |
| " | 00200 | 0215 | " |
| " | 00201 | 0215 | " |
| " | 00201 | 0212 | " |
| "(FORM FEED) | 00203 | 0214 | " (FORM FEED) |
| "(TAB) | 00204 | 0211 | "(SPACES TO NEXT TAB STOP) |

TEXT STRINGS

A string of text characters can be entered by giving the pseudo-op TEXT followed by a single space (code 240), any delimiting character (except blank tape or leader-trailer), a string of text, and the same delimiting character. For example:

```
TEXT ATEXTA
```

The character codes are stored two per word in ASCII code that has been trimmed to the rightmost six bits. Following the last character, a 6-bit zero is inserted as a stop code. The above statement would produce:

```
2405
3024
0000
```

The string in the following example:

```
TEXT /.!./
```

would produce:

```
5641
56ØØ
```

The TEXT pseudo-op could also be used as part of a calling sequence to a subroutine:

Example 1:

```
JMS MESS
TEXT /  /
```

Example 2:

```
        JMS MESS
        NOWDS           /NO WORDS IN MESSAGE
        ADDMES          /ADDRESS OF MESSAGE
        .
        .
ADDMES, TEXT /MESSAGE/
```

**NOTE**

While the TEXT pseudo-op causes characters to be stored in a trimmed code, the use of the single character control (") causes characters to be stored as a full 8-bit ASCII code.

With regard to delimiting characters, it should be noted that the Symbolic Editor always generates carriage return as carriage return/line feed and tab as tab/rubout or as multiple spaces. If carriage return (code 215) or tab (code 211) were chosen as delimiter, line feed (code 212) or rubout (code 377) would become the first character of the text string..

**NOTE**

If no delimiter is seen, line feed/dollar sign will terminate the pass. If no dollar sign is found, an effective PAUSE (on high-speed input) is executed, or $ may be typed (on low-speed input). (See HALTS at the end of this manual.)

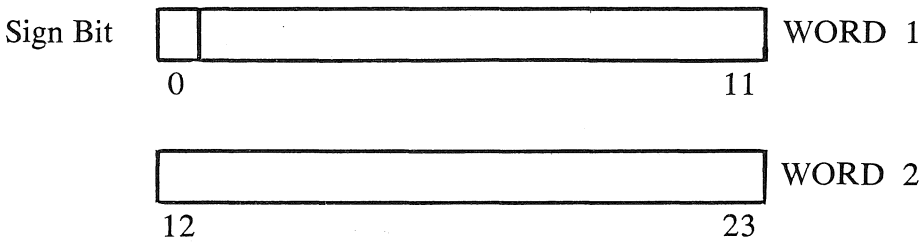
NUMBERS

The types of numbers allowed in MACRO-8 assemblies are integers, double precision integers, and double precision floating point

numbers. If the characters 8 and 9 are encountered in octal radix, they are flagged as illegal characters and ignored. Thus 1283 in octal radix becomes 123.

*Double Precision Integers*

Double precision integers may be positive or negative (stored as two's complement) according to their sign but may not be combined with operators in expressions. They are always taken as decimal radix although the current radix of the program is not disturbed. Each double precision integer is allotted two consecutive words with the sign indicated by bit 0 of the first word, as shown below:




Double precision constants must be in the range:

$$-8388608 \leqslant N \leqslant 8388607$$

The double precision integer mode is entered through the use of the pseudo-op DUBL. All numbers encountered after the occurrence of DUBL are considered double precision integers (stored in 2 words) until an alphabetic character, *, or $ is encountered. Each number is terminated by a carriage return, semicolon (;) or comment. For example:

```
*400
DUBL    679467
        44
        -3
TAG,    CLA
        .
        .
```

Once the double precision mode has been entered, the programmer must terminate it with an alphabetic character, *, or $ before proceeding with the program. With the exception of unique floating point format characters (. and E), the same rules of character

recognition apply for double precision integers as for floating point constants.

The preceding section of code would produce:

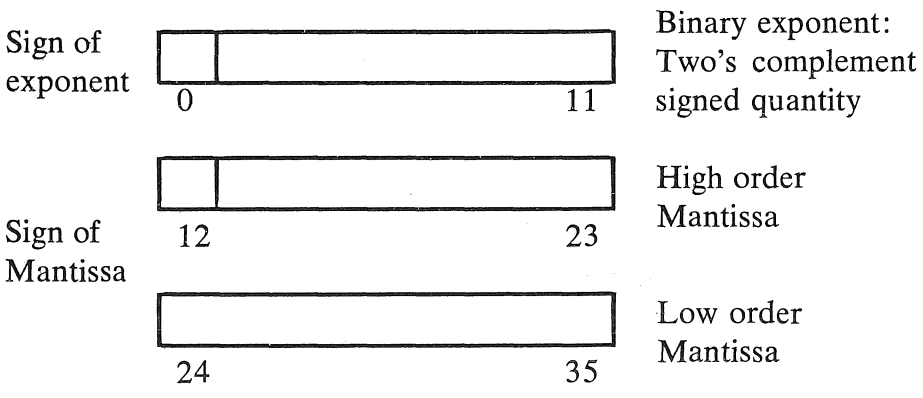
| Location | Contents |
|---|---|
| 00400 | 0245 |
| 00401 | 7053 |
| 00402 | 0000 |
| 00403 | 0054 |
| 00404 | 7777 |
| 00405 | 7775 |
| 00406 | 7200 |

The numbers indicated under contents are the octal equivalent of the decimal numbers in the preceding example. The CLA instruction is given the value 7200 as found in the permanent symbol table. The symbol TAG would have a value of 0406.

*Floating Point Constants*

The floating point input facility is designed to assemble constants for use by the PDP-8 Floating Point Package.

Double precision floating point constants may be positive or negative according to their sign but cannot be combined with operators. Decimal radix is assumed but the current radix of the program is not altered. Floating point constants are each assigned three words and are stored in normalized form, as shown below:




The exponent is a signed two's complement quantity in one 12-bit word. The signed two's complement mantissa is stored in two 12-bit words, maintaining 23 bits of significance, making a total of three words for storage.

The double precision floating point mode is entered through use of the pseudo-op FLTG. All numbers encountered after the use of FLTG will be interpreted as double precision floating point constants until the occurrence of an *, a $, or an alphabetic character other than E. The general input format of a floating point number is:

DDD.DDDEDD

where each D is a decimal digit. While in floating point mode, each character is handled in one of the following ways:

1. If it is a legal part of the format of a floating point number, it is handled as such and is used to generate code.
2. If it is an asterisk, a dollar sign, or an alphabetic character other than E, it terminates the current number and terminates floating point or double precision conversion.
3. If it is a format character (i.e., space, slash, carriage return), it terminates the current number and is otherwise ignored. Comments are ignored.
4. If it does not fall into any of the above categories, i.e., = , (%) an IC error message is generated. The number is terminated, and the character is ignored.

In floating point format, the mantissa and the exponent may be preceded by signs (+ or −). The decimal point and the exponent may be omitted. For example:

```
*400
FLTG     +509.32E02
         -62.97E-4
         1.00E-2
TAG,     CLA
```

would produce upon execution:

| Location | Contents |
|---|---|
| 00400 | 0003 |
| 00401 | 2427 |
| 00402 | 6675 |
| 00403 | 7771 |
| 00404 | 4615 |
| 00405 | 2172 |

| Location | Contents |
|---|---|
| 00406 | 7772 |
| 00407 | 2436 |
| 00410 | 5604 |
| 00411 | 7200 |

and the symbol TAG would be assigned a value of 0411.

Accuracy of floating point conversion compares favorably with the PDP-8 Floating Point Package. The most significant variation is in the handling of large negative exponents. These are calculated by multiplying by an inverse rather than by dividing, due to space limitations. No error checking is done in the floating point routines; users, therefore, should stay within the range of 7 significant digit mantissas and exponents in the range −600 to 600.

**User Defined Macros**

When writing a program, it often happens that certain coding sequences are used several times with different arguments. If so, it is convenient to generate the entire sequence with a single statement. To do this, the coding sequence may be defined as a "macro," using dummy arguments. A single statement referring to the macro name, along with a list of real arguments, will generate the correct sequence in line with the rest of the coding.

## DEFINING A MACRO

The macro name must be defined before it is used. The macro is defined by means of the pseudo-op DEFINE followed by the macro name and a list of dummy arguments separated by spaces. For example, a simple macro to move the contents of word A to word B and leave the result in the accumulator could be coded as follows:

```
DEFINE MOVE DUMMY1 DUMMY2
<CLA
TAD DUMMY1
DCA DUMMY2
TAD DUMMY2>
```

The choice of symbols used as dummy arguments is arbitrary; however, they may not be defined or referenced prior to the macro definition. The actual coding of the macro is enclosed in angle brackets.

The preceding definition of the macro MOVE can also be written as follows:

```
DEFINE MOVE ARG1 ARG2
<CLA;TAD ARG1;DCA ARG2;TAD ARG2>
```

The definition of the macro is enclosed in angle brackets, as mentioned before and the semicolon characters indicate the termination of a line of code, as in PAL III.

When a macro name is processed by the Assembler, the real arguments replace the dummy arguments. For example, assuming that the macro MOVE has been defined as above:

```
*400
A,      0
B,      -6
        MOVE A,B
```

the following code is produced:

| Location | Contents |
|---|---|
| 00400 | 0000 |
| 00401 | 7772 |
| 00402 | 7200 |
| 00403 | 1200 |
| 00404 | 3201 |
| 00405 | 1201 |

Notice that a macro definition has spaces separating the dummy arguments and the macro call has commas separating the macro arguments.

A macro need not have any arguments. For example, a sequence of coding to rotate the accumulator and link six places to the left might be coded as a macro by means of the following code:

```
DEFINE ROTL
<RTL;RTL;RTL>
```

If core space is tight and the sequence is to be used several times, it would be better to code this as a subroutine:

```
ROTL,   Ø
        RTL
        RTL
        RTL
        JMP I ROTL
```

The subroutine occupies five locations and each call, JMS ROTL, occupies 1. Each call to the macro ROTL occupies three locations. The main advantage of the macro is the use of dummy arguments. Its greatest disadvantage is the amount of symbol space required.

The entire macro definition is placed in the macro table, two characters per word, with a dummy argument value replacing the symbolic names. For example:

```
DEFINE LOAD A
<CLA
TAD A>
```

is stored, in the macro table, roughly as follows:

|CL|A |TA|D |7700|>00

where the vertical lines indicate successive 12-bit words. Comments and line feeds are not stored. The macro definition can consist of any valid coding, with the following restrictions:

1. Macros cannot be nested, i.e., another macro name or definition cannot appear in a macro definition and cannot be brought in as an argument at the time a macro is referenced.
2. TEXT (or " type) statements cannot appear in a macro definition.
3. Arguments cannot be another macro name, a TEXT pseudo-op or a " character.
4. The symbols used as dummy arguments must not have been previously defined or referenced or subsequently redefined.
5. A macro cannot be redefined (the macro name may not be used for any other purpose).
6. A macro definition must end with a >, which may not occur in a comment (comments are ignored).
7. Dummy arguments may neither be pseudo-ops nor contain pseudo-ops, and the real arguments are subject to the same restriction.

The programmer who wishes to use pseudo-ops in a macro definition (not as arguments) should experiment with short programs using the intended macro to be sure that its expansion and execution are legal and as expected or desired.

Consider the following macro definition:

```
DEFINE LOOP A B
        <TAD A
        DCA B
        TAD COUNT
        ISZ B
        JMP .-2>
```

A macro is referenced by giving the macro name, a space and the list of real arguments, separated by commas. There must be at least as many arguments in the macro call as in the corresponding macro definition. When a macro is referenced, its definition is found, expanded, and the real arguments replace the dummy arguments. The expanded macro is then processed in the normal fashion. For example, the macro call:

```
LOOP X,Y2
```

in the context of the program in which it appears, is equivalent to:

```
TAD X
DCA Y2
TAD COUNT
ISZ Y2
JMP .-2
```

The macro table shares the available space with the symbol table (see Symbol Table). Thus the programmer must be aware of the amount of room required by his macros and the fact that each symbol occupies four words of memory. Also, the arguments of a macro call are temporarily stored in this buffer space while the macro is being expanded.

**MACRO-8 Pseudo-Operators**

The following list summarizes the pseudo-ops available in MACRO-8.

| Pseudo-op | Description |
|---|---|
| DECIMAL | All following numbers interpreted as decimal |
| DEFINE | Used to define a macro |
| DUBL | Enter double precision integer mode |
| EXPUNGE | Deletes permanent symbol table with exception of pseudo-ops |
| FIELD | Causes the literals to be dumped and a field setting to be output |
| FIXTAB | Appends all presently defined symbols to the permanent symbol table |
| FLTG | Enter double precision floating point mode |
| I | Indirect addressing |
| OCTAL | Resets radix to its original octal base |
| PAGE | Resets the location counter |
| PAUSE | Signals assembler to stop processing paper tape being read |
| TEXT | Allows a character string to be entered |
| Z | Ignored (convenience to programmer only—see PAL III) |
| $ | End-of-pass (MACRO-8 recognizes line feed followed by $ as unconditional end-of-pass, whether in a text string, macro definition, or anywhere.) |

**Symbol Table**

Because of the extra features of MACRO-8, there is less room available for symbol storage than was found in PAL III. Programs that were originally coded to be assembled by PAL III may have too many symbols to be assembled by MACRO-8. If register switches 9 and 10 are set to 1 during assembly, MACRO-8's user symbol table will be extended. These switches cause the macro processor and double precision and floating point processor to be deleted (see the operating instructions).

The high-speed reader buffer occupies $400_8$ locations which will be used for symbol storage if low-speed input is requested. With high-speed input, the symbol table capacity is $115_{10}$ symbols, each of which requires four core locations. Using the LT-33 as the input device, the capacity is $179_{10}$ symbols.

## SYMBOL TABLE MODIFICATION

Because of the small amount of core remaining to be used for programmer symbols and the macro table, the following suggestions are offered allowing a particular installation or individual to conserve symbol table space.

By use of the pseudo-ops EXPUNGE and FIXTAB, unnecessary instruction mnemonics can be removed from the symbol table, making more space available for programmer defined symbols and macros. This also decreases assembly time as the unused instruction symbols are not involved in the symbol table searches. The most often used instruction mnemonics should be assembled first, so that they will be in core next to the special characters and pseudo-ops. This is desirable because the symbol search routine starts with the pseudo-ops at the top of the table (7577) and works down.

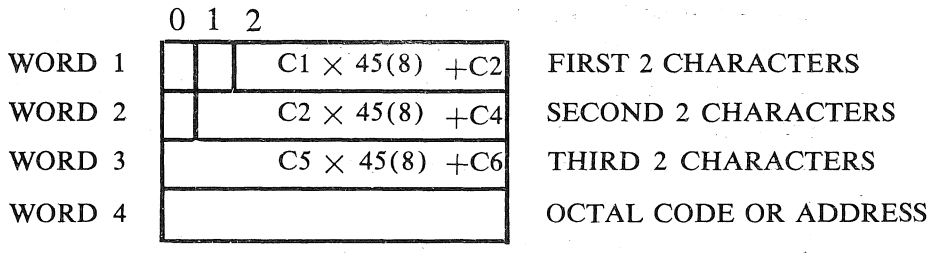
At an installation which does not have optional equipment (RF08 disk, TC01 DECtape, high-speed reader/punch, etc.) available, the corresponding instruction sets can be removed. A symbolic tape beginning with EXPUNGE, containing all necessary instruction mnemonic definitions, and ending with FIXTAB and $ could be assembled (only pass 1 is necessary) by MACRO-8 prior to any other assemblies. For example:

```
EXPUNGE
AND=0000
TAD=1000
CLA=7200
.
.
FIXTAB
$
```

The pseudo-op PAUSE could also be used in place of the dollar sign with the above tape, as the first tape of a multiple tape assembly. The definitions will remain in the table until a subsequent EXPUNGE or until the Assembler is reloaded. See the list of permanent symbols in Appendix C.

## INTERNAL SYMBOL REPRESENTATION FOR MACRO-8

Each permanent user defined symbol occupies four words (locations) in the symbol table storage area, as follows:




where C1, C2, . . . , C6 represent the first character, second character, . . . , sixth character, respectively. (Symbols may consist of from one to six characters.) Bits 0 and 1 of word 1, and bit 0 of word 2 are a three digit code denoting the type of symbol. Word 4 contains the octal code of the symbol; if a user defined symbol, Word 4 contains the address of the symbol. For example, the permanent symbol TAD is represented as follows:

WORD 1 = $24_8 \times 45_8 + 01 = 1345_8$ OR TA
WORD 2 = $04_8 \times 45_8 + 00 = 224_8 + 4000_8 = 4224_8$ OR D
WORD 3 = 0000
WORD 4 = 1000 (OCTAL CODE FOR TAD)

Note that the octal code for each character is always scaled by the Assembler so that the character is represented using six bits of a word. For example, ASCII code for T is 324, and was trimmed to 24; A is 301, trimmed to 01; etc. Digits 0 through 9 are scaled to the range 33 through 44.

MACRO-8 recognizes eight categories of symbols:

1. Special Characters
   These include single non-alphanumeric characters which have special meaning to the Assembler, i.e., + − , = !. They are not affected by EXPUNGE or FIXTAB and may not be redefined. The programmer may not define additional special characters. Special characters are recognized by the Assembler by their three digit code of 011.
2. Pseudo-ops
   These are instructions to the Assembler, which, with the exception of FIELD, do not produce binary code. When they are recognized by the Assembler, they are executed immediately. They are not affected by EXPUNGE or FIXTAB and may not be redefined. The programmer may

not define additional pseudo-ops. Pseudo-ops may not occur to the left of an = sign or , (comma) nor may pseudo-ops other than I and Z in their proper context appear to the right of an = sign. Pseudo-ops are recognized by the Assembler by their three digit code of 100.

3. Permanent Symbols
   These are symbols which are part of the permanent assembler symbol table or which were used in the program prior to the pseudo-op FIXTAB. They include such symbols as TAD, CLA, TLS, and any user-defined permanent symbols. They are designated as permanent symbols by their position in the table and are not necessarily defined. For example, the coding sequence

```
TAD A
FIXTAB
```

   will cause A to be entered as a permanent symbol with no definition. It must later be defined, at which time it will be the same as any other permanent symbol. Permanent symbols may be redefined only if the new definition agrees with the old.
4. User-defined symbols
   These include all non-permanent, non-pseudo-op, non-special character symbols. They are recognized by the Assembler by their place in the symbol table.
5. Defined symbols
   These are permanent or user-defined symbols which either are part of the assembler symbol table or have been defined with = or a comma. Permanent symbols and those defined with comma may only be redefined if the new definition agrees with the old. Any user-defined symbol may be redefined with =. They are recognized by the Assembler by the three digit code 001.
6. Undefined symbols
   These have never been defined with = or a comma. They are stored in the symbol table in the order of their occurrence and are recognized by the Assembler by their three digit code of 010.
7. Macro names
   These are the names defined with the pseudo-op DEFINE

as the names of macros. They are stored with the user-defined and permanent symbols and are recognized by the Assembler by their three digit code of 000.

8. Macro dummy arguments
   These are the arguments which follow the macro name in the macro definition. These are stored with the permanent and user-defined symbols and are recognized by the Assembler by their three digit code of 101.

## MEMORY REFERENCE INSTRUCTION RECOGNITION

Memory reference instructions are recognized by their use in an expression, according to the following rule:

Given an expression SYM1 SYM2, SYM1 is a memory reference instruction if and only if SYM1 is a permanent symbol and SYM2 is not a permanent symbol. In any other case SYM1 and SYM2 will be combined with an inclusive OR. For example:

| | |
|---|---|
| CLA CLL | Both permanent; will be combined with inclusive OR. |
| A B | Both user-defined; will be combined with inclusive OR. |
| B CLA | User-defined followed by permanent; will be combined with inclusive OR. |
| AND A | Permanent symbol followed by user-defined; permanent symbol will be treated as a memory reference instruction and user-defined symbol as referenced address. |

## COMPATABILITY BETWEEN PAL III AND MACRO-8

MACRO-8 will assemble code produced for PAL III with the following exceptions:

1. The symbol table in MACRO-8 is considerably shorter than the symbol table in PAL III due to the added features of MACRO-8. See the section dealing with limited symbol space for suggestions on breaking large tapes into smaller ones.
2. MACRO-8 has no FIXMRI pseudo-op. Refer to the section on internal symbol representation for a discussion of MACRO-8 memory reference instruction recognition.
3. Tags which are legal user-defined symbols in PAL III,

such as PAGE, DUBL, or FLTG, may be pseudo-ops in MACRO-8.

4. While PAL III allows two levels of definition with =, i.e., A=B=0, MACRO-8 only allows one level, A=0; B=0.
5. A special check was inserted into PAL III for the convenience of programmers who forgot the space in code involving ., (such as JMP .−1); no such check exists in MACRO-8.

**Programming Hints**

1. Arrange coding in order of ascending address to avoid confusion.
2. Comment profusely wherever possible.
3. All terms in any origin expression (* or PAGE) need to be defined prior to that expression so that all terms defined thereafter will be given the proper value on pass 1.
4. All terms to the right of an equal sign (=) must have been previously defined so that the term on the left is given the proper value on pass 1.
5. Since line feed/dollar sign is recognized as unconditional end-of-pass, it is wise to develop the habit of typing the terminating $ at the left hand margin so that it will be caught if a TEXT or macro delimiter is missing. It is also wise to keep several short tapes with just carriage return/line feed, dollar sign, carriage return/line feed in the tape tray for assembling short subroutine tapes which end in PAUSE. Often these subroutines may be debugged separately prior to inclusion in a larger program.
6. If the link generating facility is used, it may be wise to suppress the LG error message on pass 2. In MACRO-8 (version DEC-08-CMAB-PB) location 1233 contains 4777 normally. To suppress the LG message, reset location 1233 to contain 7200 (CLA) on pass 2. It should be reset to 4777 on pass 3.

## DEALING WITH A LIMITED SYMBOL SPACE

1. Breaking long programs into smaller ones

   Due to a shortage of symbol space it is often necessary to break a large program into several smaller ones and assemble each one separately. Then the binaries may all be

loaded into core together. Record separately the map of where each program fits into core, so that two segments are not assembled and loaded into the same area.
Make each segment as self-sufficient as possible. At the beginning of each tape, insert comments indicating where that segment expects other segments to be and define tags for locations which must be referenced. For example, a subroutine which calculates the sine of a number starts at *400. It expects the double precision multiplication subroutine to be at *200.
The beginning of the multiply subroutine:

```
/DCUBLE PRECISION MULTIPLY SUBROUTINE
/CALLED WITH ADDRESS OF HIGH ORDER MULTIPLIER
/IN AC AND ADDRESS OF HIGH ORDER MULTIPLICAND
/IN LOCATION FOLLOWING JMS
/ FCR EXAMPLE:
/       TAD     MULT1
/       JMS     MULT
/       MULT2
*200
MULT,   0
```

The beginning of the sine subroutine:

```
/DOUBLE PRECISION SINE SUBROUTINE
/EXPECTS DOUBLE PRECISION MULTIPLY SUBROUTINE
/TO START AT LOCATION 200
/CALLED WITH ADDRESS OF ARGUMENT IN AC
/STORES RESULT IN LOCATIONS 500 AND 501

MULTP=200
*400
SINE,   0
        .
        .
        TAD ARG
        JMS I MULTI
        ARG2
        .
        .
MULTI,  MULTP
```

Each subroutine is self-sufficient and may be debugged and assembled separately. It is only necessary to be sure

that each piece is where it is expected to be. Comments in the source listing are a great advantage here.

2. Reducing the number of symbols in the program
   To decrease the number of symbols in a program, consider the following:
   a. Use of . instead of several tags close together, i.e., JMP .+3 instead of JMP TAG. There is a reasonable limit, around $10_8$ instructions, beyond which a separate tag is more feasible. A JMP .+17 can be trouble if a line of code is removed later.
   b. Lists of pointers and constants instead of scattering them through a page, for example:

```
LIST,   TAG1
        TAG2
        TAG3
        TAG4
```

   rather than:

```
START,  TAG1
.
.
END,    TAG2
.
.
GO,     TAG3
```

   These locations may then be referenced as follows:

```
JMS I LIST+2
```

   instead of:

```
JMS I GO
```

   c. Let the Link Generating Routine handle the reference:

```
JMS TAG3
```

   d. If links are not wanted, make use of the Literal Processor to free symbol space:

The effect of c and d is the same, with the exception of the diagnostic message LG (Link Generated). (Many programmers avoid generating links so that if one is generated, it will be noted as an error.)

3. Use of Switch Options (see MACRO-8 Switch Options)
   If a program does not use macros or floating point, deleting those processors will provide more symbol space. If the program is very long and will only need to be assembled once, using the low speed reader instead of the high speed reader gains 64 symbols worth of space.
4. Revising symbol table
   If a program does not require all of the Assembler's permanent symbols, the extra ones may be deleted by using EXPUNGE, redefining the necessary ones, and using FIXTAB. This gains one symbol space for each mnemonic deleted. (See Appendix C for the permanent symbol table.) If the EXPUNGE definition and FIXTAB are put on a separate tape, this tape need only be assembled on pass 1.
5. Curtail the use of macros wherever possible
   Use of a subroutine in place of a macro will decrease the amount of symbol space necessary since the macro is stored in its entirety in the symbol table.
6. Punching a new assembler tape
   If MACRO-8 is to be used frequently with a revised symbol table, it may be wise to punch a new tape of the Assembler with the revised table (and deleted processors, if any). To do so, it is necessary to obtain the source tapes for the Binary Punch Routine (DEC-08-YX1A-PB), edit it to reset the origin to 6000, and punch a new binary tape. (The Binary Punch Routine currently resides in core in the same locations as MACRO-8's permanent symbol table.) Load MACRO-8, EXPUNGE, redefine, and modify as necessary. Then load the new Binary Punch Routine and cause it to dump all core from locations 0 to 7577. The resulting tape is the new Assembler. (Location 6000 is in the middle of the symbol table and using it for the origin should allow plenty of room for the Binary Punch

Routine. If there is doubt, obtain the MACRO-8 source listing for reference.)

**Summary of MACRO-8 Error Diagnostics**

The format of the error messages is:

ERROR CODE ADDRESS

where ERROR CODE is a two character code which specifies the type of error, and ADDRESS is either the absolute octal address where the error occurred or the address of the error relative to the last symbolic label (if there was one) on that page.

Assembly can be continued after most errors. On nonrecoverable errors, assembly halts, and the program must be edited to correct the error and assembly restarted.

**Table 1 MACRO-8 Error Messages**

| Error Code | Meaning |
|---|---|
| BE | Two MACRO-8 internal tables have overlapped. This situation can usually be corrected by decreasing the number of current page literals used prior to this point on the page. It is sometimes the result of too many nested literals in an expression. |
| IC | Illegal Character<br>1. A non-valid character was found other than in a comment field or a text field<br>2. A valid character was found under the wrong conditions, i.e., 8 or 9 in octal radix<br>The illegal character is flagged and ignored, subject to conditions stated elsewhere in this chapter. (See floating point constants, double precision integers, etc.) |
| ID | Illegal redefintion of a symbol<br>An attempt was made to give a previously defined symbol a new value. The symbol was not redefined (this is similar to the duplicate tag diagnostic of PAL III). |
| IE | Illegal Equals<br>An equal sign was used in the wrong context. For example:<br>`TAD=Ø33`<br>`A+B=C`<br>`A=B=Ø` |

**Table 1 (Con't) MACRO-8 Error Messages**

| Error Code | Meaning |
|---|---|
| | The expression to the left of the equal sign must be a single symbol.<br>Only one level of definition is allowed in MACRO-8 (unlike PAL III where two levels are allowed). Permanent symbols may be redefined only if the new and old definitions match. (After EXPUNGE, only pseudo-ops remain.) User-defined symbols may be redefined with = at any time. |
| II | Illegal Indirect<br>An out-of-page reference was made, and a link could not be generated because the indirect bit was already set. For example:<br><br>`*200`<br>`TAD I A`<br>`.`<br>`.`<br>`PAGE`<br>`A,  CMA CLL` |
| IM | Illegal format in a Macro definition<br>The expression after the DEFINE pseudo-instruction does not comply with the macro definition position or structural rules. For example: a macro name is referenced before the macro definition. |
| LG | Link Generated<br>A warning message; a link was generated for an out-of-page reference at this address. For example:<br><br>`          *200`<br>`LG 00200  0177      TAD A   /COMMENT`<br>`                .`<br>`                .`<br>`  00377  0400`<br>`          PAGE`<br>`  00400  7140  A,    CLA CLL` |
| MP | Missing Parameter in a macro call<br>An argument, called for by the macro definition, is missing. For example: |

**Table 1 (Con't) MACRO-8 Error Messages**

| Error Code | Meaning |
|---|---|

```
                        DEFINE MAC A B
                                <TAD A
                                CIA
                                DCA B>

                        *200
                        SUM=300

    00200   1300               MAC SUM
    00201   7041
MP  00202   3000
```

PE — Current, non-zero Page Exceeded
An attempt was made to override a literal with an instruction, or override an instruction with a literal. This can be corrected by decreasing the number of literals on the page, or decreasing the number of instructions on the page.

SE — Symbol Table Exceeded
The symbol table and macro table overlap. Non-recoverable error. The number of symbols or macros must be reduced.

US — Undefined Symbol
A symbol has been processed during pass 2 that was not defined by the end of pass 1.

XX — Reference was made to a deleted processor through one of the following symbols: DUBL, FLTG, DEFINE, <, >. Non-recoverable error. The Assembler must be reloaded to handle the source tape or the source tape must be edited to remove the illegal reference.

ZE — Page zero exceeded. Same as PE only with reference to page 0.

**Pass 3 Output—Assembly Listing**

The output on pass 3 is a side-by-side listing of the source program and its generated octal code. Depending on the setting of bit 11, this output is to either the Teletype or the high-speed punch. The listing has the following characteristics:

The first 3 characters of the line are reserved for a two character error message, if necessary, and a single space. If there is no error message for that line, the spaces are left blank. If

there is more than one error message, the succeeding ones will be added to the first with no intervening spaces and the remainder of the line will be shifted to the right.

The 4th through 8th characters contain the address into which coding on that line was assembled. If there was no assembled coding, these characters are spaces. The address is a four digit octal address preceded by a one digit field number.

The 9th and 10th characters are spaces.

The 11th through 14th characters contain the assembled code, if any. If there was none, these characters are also spaces.

The 15th and 16th characters are spaces.

The 17th through 72nd characters contain the source coding, including comments. Any tab characters (ASCII code 211) appearing in the source code will be output as spaces to the next tab stop. Tab stops occur every eight spaces in the source code field. Since an LT-33 has a 72 character line length, there is only room for a total of 56 characters in the source code and comment on one line. Any extra characters will be typed over the last of the 56 characters. For neat listings on the LT-33, comments should be restricted so that the line does not exceed 56 characters. If the listing is to be generated on the high-speed punch for later printing on a wider device, this restriction can be modified without changing the Assembler.

Since the LT-33 does not handle a form feed (ASCII code 214) as a top-of-form, MACRO-8 will output this character as six blank lines, a form feed, and six blank lines, so that listings may be conveniently divided into pages.

A sample of this format is given under the LG error message in the previous section.

Dollar sign is not printed when it is the terminating pseudo-op. (It is printed in a text string or single character text.)

**MACRO-8 Operating Procedures**

ASSEMBLER OUTPUT

MACRO-8 is a two pass assembler with an optional third pass which produces an octal/symbolic assembly listing. During the first pass, MACRO-8 processes the source tape and places all symbol definitions and macro definitions in its symbol table and macro

table, respectively. During the second pass, MACRO-8 processes the source tape and punches the binary format tape and symbol table. This punched table can be read by DDT. The third pass provides a listing of the generated octal code and the original source language followed by a printed symbol table.

*Input Device*

MACRO-8 can be used with either the high-speed reader or with the low-speed ( LT-33 ) reader. The choice is determined on first pass by the location of the source tape and is remembered for the other two passes.

*Output Device*

On pass 1 the only output is to the Teletype in the form of the error messages which have been generated.

On pass 2 the device on which the binary tape is to be punched is determined by bit 11 of the switch register. If bit 11 is set to 0, the LT-33 will be used. If bit 11 is set to 1, the high-speed punch will be used. The punch should be turned on before beginning the pass. The symbol table will be punched on the same device (also printed if the LT-33 was specified.) Any error messages will appear on the LT-33.

On pass 3, the same choices on bit 11 will determine whether the listing will be produced on the Teletype or the high speed punch.

## OPERATING INSTRUCTIONS

1. Loader (see Appendix A). Leave the data field and instruction field set to this field for the remainder of the assembly. (On a 4K machine, load into field 0.)
2. Prepare for pass 1.
   a. Turn Teletype to LINE
   b. Place source in reader
   c. Turn on reader
   d. Set switch register to 0200
   e. Press ADDRess LOAD
   f. Set switch register for pass and desired options (see MACRO-8 Switch Options)
   g. Press CLEAR and CONTinue

   If there are multiple tapes, each ending in PAUSE, when the computer halts, remove the tape from the reader, insert the

next tape, and press CONTinue. Repeat until all tapes have been read.

3. Passes 2 and 3
   a. Place source tapes in reader (if not the same reader as the previous pass, follow instructions from b under pass 1).
   b. Set switch register for pass and desired option. (see MACRO-8 Switch Options)
   c. Turn on output device, if not already on.
   d. Press CONTinue.

If there are multiple tapes, follow the procedure for them as given under pass 1.

MACRO-8 SWITCH OPTIONS

Switches should be set as noted to obtain the desired option. Switches not mentioned are ignored by the Assembler.

PASS SETTING

| Bit 0 | Bit 1 | Option |
|---|---|---|
| 0 | 0 | Pass 1 |
| 1 | 1 | Pass 1—to retain all previously defined symbols |
| 0 | 1 | Pass 2 |
| 1 | 0 | Pass 3 |

*Processor Deletion*

Certain processors may be deleted to make room for more symbols:

Bit 9 = 1 Delete the floating point and double precision processors. This increases the size of the symbol table by $68_{10}$ symbols.

Bit 10 = 1 Delete the macro processor and the floating point and double precision processors. This increases symbol space by $131_{10}$ symbols.

**NOTE**

Switches 9 and 10 are sensed whenever pass 1 is entered. Once processors have been deleted, MACRO-8 must be reloaded to handle subsequent programs that use macros, double precision integers, or floating point numbers. Reference to a deleted processor is a non-recoverable error.

*Output Device*

Bit 11 = 0 Teletype printer (and punch, if turned on)
Bit 11 = 1 High-speed punch

Error messages will be output to the Teletype, which must be turned on-line.

*Halts*

Locations mentioned in the following section refer to the version of MACRO-8 released in March 1971 (DEC-08-CMAB).

The halt on $ is at location 4232. Next pass can be started from this halt.

The halt on PAUSE is at location 1747. Another tape may be added to the current pass from this halt.

The halt on end-of-tape in high-speed reader (an effective pause) is at location 340. This would occur if no $ or PAUSE was seen. Insert separate tape of $ and press CONTinue to end pass.

If the LT-33 runs out of tape without encountering a $ or PAUSE, it stops reading and leaves the machine in an IOT loop. (KSF; JMP .−1;). Type $ on keyboard to end pass.

Other halts should be preceded by an error message.

# appendix a
# loading procedures

**Initializing the system**

Before using the computer system, it is good practice to initialize all units. To initialize the system, ensure that all switches and controls are as specified below.

1. Main power cord is properly plugged in.
2. Teletype is turned OFF.
3. Low-speed punch is OFF.
4. Low-speed reader is set to FREE.
5. Computer POWER key is ON.
6. PANEL LOCK is unlocked.
7. Console switches are set to 0.
8. SING STEP is not set.
9. High-speed punch is OFF.
10. DECtape REMOTE lamps OFF.

The system is now initialized and ready for your use.

**Loaders**

READ-IN MODE (RIM) LOADER

When a computer in the PDP-8 series is first received, it is nothing more than a piece of hardware; its core memory is completely demagnetized. The computer "knows" absolutely nothing, not even how to receive input. However, the programmer can manually load data directly into core using the console switches.

The RIM Loader is the very first program loaded into the computer, and it is loaded by the programmer using the console

switches. The RIM Loader instructs the computer to receive and store, in core, data punched on paper tape in RIM coded format (RIM Loader is used to load the BIN Loader described below.)

There are two RIM loader programs: one is used when the input is to be from the low-speed paper tape reader, and the other is used when input is to be from the high-speed paper tape reader. The locations and corresponding instructions for both loaders are listed in Table A-1.

The procedure for loading (toggling) the RIM Loader into core is illustrated in Figure A-1.

**Table A-1. RIM Loader Programs**

| | Instruction | |
|---|---|---|
| Location | Low-Speed Reader | High-Speed Reader |
| 7756 | 6032 | 6014 |
| 7757 | 6031 | 6011 |
| 7760 | 5357 | 5357 |
| 7761 | 6036 | 6016 |
| 7762 | 7106 | 7106 |
| 7763 | 7006 | 7006 |
| 7764 | 7510 | 7510 |
| 7765 | 5357 | 5374 |
| 7766 | 7006 | 7006 |
| 7767 | 6031 | 6011 |
| 7770 | 5367 | 5367 |
| 7771 | 6034 | 6016 |
| 7772 | 7420 | 7420 |
| 7773 | 3776 | 3776 |
| 7774 | 3376 | 3376 |
| 7775 | 5356 | 5357 |
| 7776 | 0000 | 0000 |

After RIM has been loaded, it is good programming practice to verify that all instructions were stored properly. This can be done by performing the steps illustrated in Figure A-2, which also shows how to correct an incorrectly stored instruction.

When loaded, the RIM Loader occupies absolute locations 7756 through 7776.

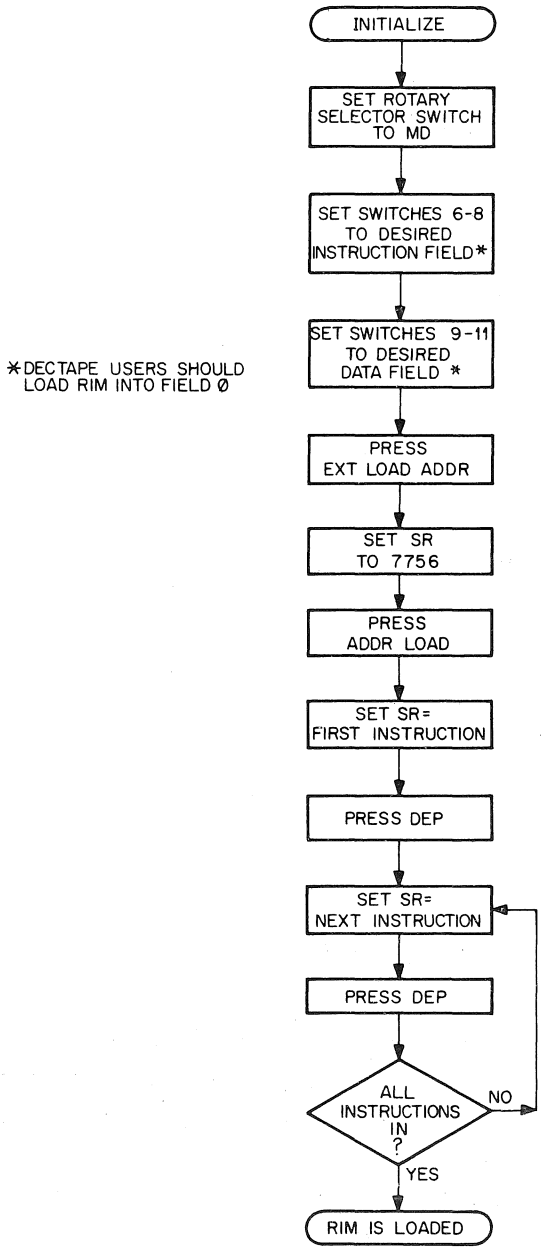



Figure A-1. Loading the RIM Loader

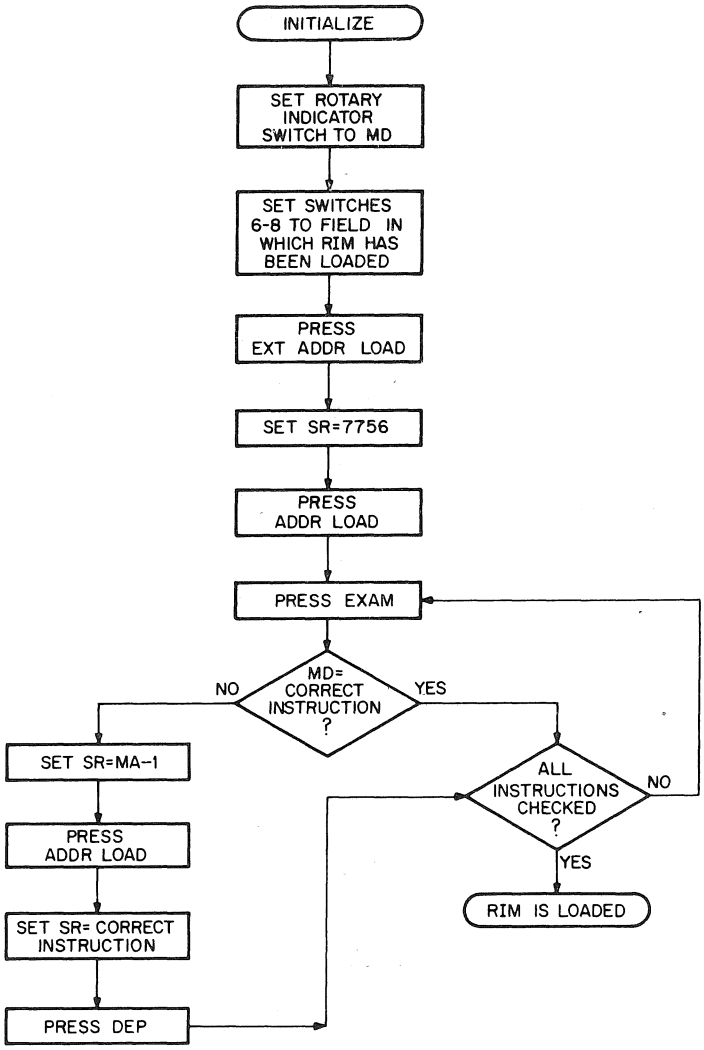



Figure A-2. Checking the RIM Loader

BINARY (BIN) LOADER—

The BIN Loader is a short utility program which, when in core, instructs the computer to read binary-coded data punched on paper tape and store it in core memory. BIN is used primarily to load the programs furnished in the software package (excluding the loaders and certain subroutines) and the programmer's binary tapes.

BIN is furnished to the programmer on punched paper tape in RIM-coded format. Therefore, RIM must be in core before BIN can be loaded. Figure A-3 illustrates the steps necessary to properly load BIN. And when loading, the input device (low- or high-speed reader) must be that which was selected when loading RIM.

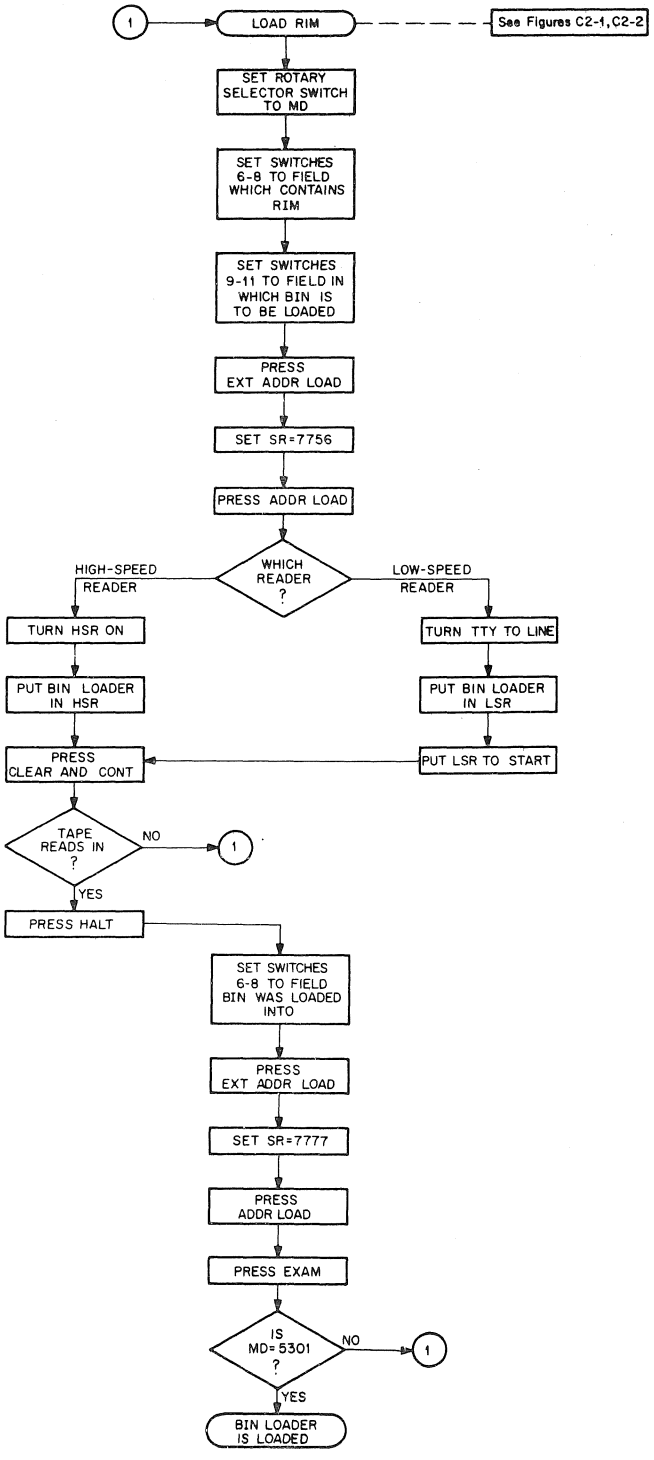



Figure A-3 . Loading the BIN Loader

When stored in core, BIN resides on the last page of core, occupying absolute locations 7625 through 7752 and 7777.

BIN was purposely placed on the last page of core so that it would always be available for use—the programs in DEC's software package do not use the last page of core (excluding the Disk Monitor). The programmer must be aware that if he writes a program which uses the last page of core, BIN will be wiped out when that program runs on the computer. When this happens, the programmer must load RIM and then BIN before he can load another binary tape.

Binary tapes to be loaded should be started on the leader-trailer code (Code 200), otherwise zeros may be loaded into core, destroying previous instructions.

Figure A-4 lilustrates the procedure for loading binary tapes into core.

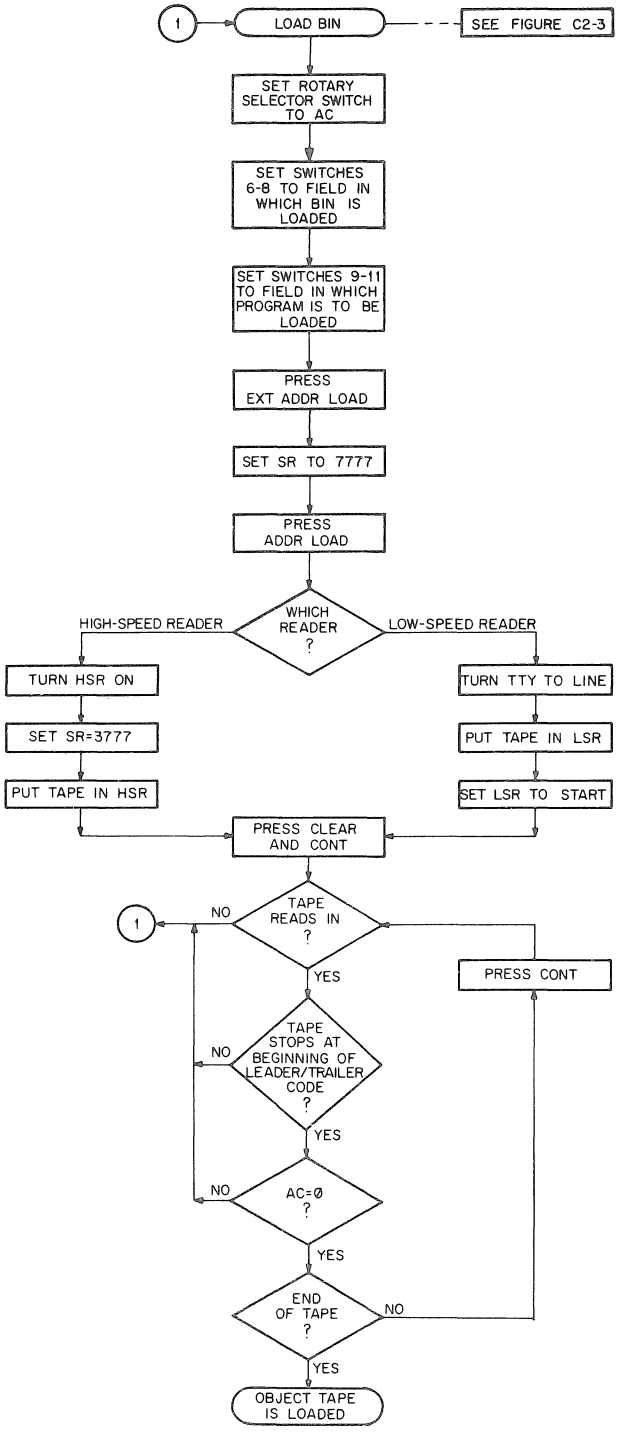



Figure A-4. Loading A Binary Tape Using BIN

# appendix b
# character codes

## ASCII-1[1] Character Set

| Character | 8-Bit Octal | 6-Bit Octal | Decimal Equivalent (A1 Format) | Character | 8-Bit Octal | 6-Bit Octal | Decimal Equivalent (A1 Format) |
|---|---|---|---|---|---|---|---|
| A | 301 | 01 | 96 | ! | 241 | 41 | −1952 |
| B | 302 | 02 | 160 | " | 242 | 42 | −1888 |
| C | 303 | 03 | 224 | # | 243 | 43 | −1824 |
| D | 304 | 04 | 288 | $ | 244 | 44 | −1760 |
| E | 305 | 05 | 352 | % | 245 | 45 | −1696 |
| F | 306 | 06 | 416 | & | 246 | 46 | −1632 |
| G | 307 | 07 | 480 | ' | 247 | 47 | −1568 |
| H | 310 | 10 | 544 | ( | 250 | 50 | −1504 |
| I | 311 | 11 | 608 | ) | 251 | 51 | −1440 |
| J | 312 | 12 | 672 | * | 252 | 52 | −1376 |
| K | 313 | 13 | 736 | + | 253 | 53 | −1312 |
| L | 314 | 14 | 800 | , | 254 | 54 | −1248 |
| M | 315 | 15 | 864 | - | 255 | 55 | −1184 |
| N | 316 | 16 | 928 | . | 256 | 56 | −1120 |
| O | 317 | 17 | 992 | / | 257 | 57 | −1056 |
| P | 320 | 20 | 1056 | : | 272 | 72 | −352 |
| Q | 321 | 21 | 1120 | ; | 273 | 73 | −288 |
| R | 322 | 22 | 1184 | < | 274 | 74 | −224 |
| S | 323 | 23 | 1248 | = | 275 | 75 | −160 |
| T | 324 | 24 | 1312 | > | 276 | 76 | −96 |
| U | 325 | 25 | 1376 | ? | 277 | 77 | −32 |
| V | 326 | 26 | 1440 | @ | 300 | | 32 |
| W | 327 | 27 | 1504 | [ | 333 | 33 | 1760 |
| X | 330 | 30 | 1568 | \ | 334 | 34 | 1824 |
| Y | 331 | 31 | 1632 | ] | 335 | 35 | 1888 |
| Z | 332 | 32 | 1696 | ↑(^)[2] | 336 | 36 | 1952 |
| 0 | 260 | 60 | −992 | ←(_)[2] | 337 | 37 | 2016 |
| 1 | 261 | 61 | −928 | Leader/Trailer | 200 | | |
| 2 | 262 | 62 | −864 | LINE FEED | 212 | | |
| 3 | 263 | 63 | −800 | Carriage RETURN | 215 | | |
| 4 | 264 | 64 | −736 | SPACE | 240 | 40 | −2016 |
| 5 | 265 | 65 | −672 | RUBOUT | 377 | | |
| 6 | 266 | 66 | −608 | Blank | 000 | | |
| 7 | 267 | 67 | −544 | BELL | 207 | | |
| 8 | 270 | 70 | −480 | TAB | 211 | | |
| 9 | 271 | 71 | −416 | FORM | 214 | | |

[1] An abbreviation for American Standard Code for Information Interchange.

[2] The character in parentheses is printed on some Teletypes.

# appendix c
# permanent symbol tables

## PAL III, MACRO-8

The following are the most commonly used elements of the PDP-8 instruction set. For that reason they are found in the permanent symbol table within most assemblers. These instructions are already defined within the computer. For additional information on these instructions and for a description of the symbols used when programming other, optional, I/O devices, see the *Small Computer Handbook,* available from the DEC Software Distribution Center.

### INSTRUCTION CODES

| Mnemonic | Code | Operation | Time ($\mu$sec.)[1] |
|---|---|---|---|
| Memory Reference Instructions | | | |
| AND | 0000 | Logical AND | 2.6 |
| TAD | 1000 | Two's complement add | 2.6 |
| ISZ | 2000 | Increment and skip if zero | 2.6 |
| DCA | 3000 | Deposit and clear AC | 2.6 |
| JMS | 4000 | Jump to subroutine | 2.6 |
| JMP | 5000 | Jump | 1.2 |
| Floating-Point Instructions | | | |
| FEXT | 0000 | Floating exit | 60 |
| FADD | 1000 | Floating add | 900 |
| FSUB | 2000 | Floating subtraction | 920 |
| FMPY | 3000 | Floating multiply | 1450 |
| FDIV | 4000 | Floating divide | 1480 |
| FGET | 5000 | Floating get | 115 |
| FPUT | 6000 | Floating put | 125 |
| FNOR | 7000 | Floating normalize | 800 |

[1] Times are representative of the PDP-8/E.

| Mnemonic | Code | Operation | Sequence |
|---|---|---|---|
| Group 1 Operate Microinstructions (1 cycle[2]) | | | |
| NOP | 7000 | No operation | — |
| IAC | 7001 | Increment AC | 3 |
| RAL | 7004 | Rotate AC and link left one | 4 |
| RTL | 7006 | Rotate AC and link left two | 4 |
| RAR | 7010 | Rotate AC and link right one | 4 |
| RTR | 7012 | Rotate AC and link right two | 4 |
| CML | 7020 | Complemented link | 2 |
| CMA | 7040 | Complement AC | 2 |
| CLL | 7100 | Clear link | 1 |
| CLA | 7200 | Clear AC | 1 |
| BSW | 7002 | Swap Bytes in AC | 4 |
| Group 2 Operate Microinstructions (1 cycle) | | | |
| HLT | 7402 | Halts the computer | 3 |
| OSR | 7404 | Inclusive OR SR with AC | 3 |
| SKP | 7410 | Skip unconditionally | 1 |
| SNL | 7420 | Skip on nonzero link | 1 |
| SZL | 7430 | Skip on zero link | 1 |
| SZA | 7440 | Skip on zero AC | 1 |
| SNA | 7450 | Skip on nonzero AC | 1 |
| SMA | 7500 | Skip on minus AC | 1 |
| SPA | 7510 | Skip on positive AC (zero is positive) | 1 |
| Combined Operate Microinstructions | | | |
| CIA | 7041 | Complement and increment AC | 2, 3 |
| STL | 7120 | Sent link to 1 | 1, 2 |
| GLK | 7204 | Get link (put link in AC, bit 11) | 1, 4 |
| STA | 7240 | Set AC to — 1 | 2 |
| LAS | 7604 | Load AC with SR | 2, 3 |
| MQ Microinstructions | | | |
| MQL | 7421 | Load MQ from AC, then clear AC | |
| MQA | 7501 | Inclusive OR the MQ with AC | |
| CAM | 7621 | Clear AC and MQ | |
| SWP | 7521 | Swap AC and MQ | |
| ACL | 7701 | Load MQ into AC | |
| Internal IOT Microinstructions | | | |
| ION | 6001 | Turn interrupt processor on | |
| IOF | 6002 | Disable interrupt processor | |

[2] 1 cycle is equal to 1.2 microseconds.

| Mnemonic | Code | Operation |
|---|---|---|
| Internal IOT Microinstructions (1 cycle) | | |
| SKON | 6000 | Skip if interrupt ON, and turn OFF |
| SRQ | 6003 | Skip on interrupt request |
| GTF | 6004 | Get interrupt flags |
| RTF | 6005 | Restore interrupt flags |
| SGT | 6006 | Skip on greater than flag |
| CAF | 6007 | Clear all flags |
| Keyboard/Reader (1 cycle) | | |
| KSF | 6031 | Skip on keyboard/reader flag |
| KCC | 6032 | Clear keyboard/reader flag and AC; set reader run |
| KRS | 6034 | Read keyboard/reader buffer (static) |
| KRB | 6036 | Clear AC, read keyboard buffer (dynamic), clear keyboard flags |
| KCF | 6030 | Clear keyboard/reader |
| KIE | 6035 | AC 11 to keyboard/reader interrupt enable F.F. |
| Teleprinter/Punch (1 cycle) | | |
| TSF | 6041 | Skip on teleprinter/punch flag |
| TCF | 6042 | Clear teleprinter/punch flag |
| TPC | 6044 | Load teleprinter/punch and print |
| TLS | 6046 | Load teleprinter/punch, print, and clear teleprinter/punch flag |
| TFL | 6040 | Set teleprinter/punch flag |
| TSK | 6045 | Skip on printer or keyboard flag |
| High Speed Reader—Type PR8/E (1 cycle) | | |
| RSF | 6011 | Skip on reader flag |
| RRB | 6012 | Read reader buffer and clear reader flag |
| RFC | 6014 | Clear flag and buffer and fetch character |
| RPE | 6010 | Set interrupt enable for reader and punch |
| RCC | 6016 | Read reader buffer, clear flag and buffer, and fetch character |
| PCE | 6020 | Clear interrupt enable for reader and punch |

| Mnemonic | Code | Operation | Time ($\mu$sec.) |
|---|---|---|---|
| High Speed Punch—Type PP8/E (1 cycle) | | | |
| PSF | 6021 | Skip on punch flag | |
| PCF | 6022 | Clear flag and buffer | |
| PPC | 6024 | Load punch buffer and punch character | |
| PLS | 6026 | Clear flag and buffer, load buffer and punch character | |
| RPE | 6010 | Set interrupt enable for reader and punch | |
| PCE | 6020 | Clear interrupt enable for reader and punch | |

| Mnemonic | Code | Operation |
|---|---|---|
| Memory Extension Control, Type MC8/E (1 cycle) | | |
| CDF | 62N1 | Change to data field N |
| CIF | 62N2 | Change to instruction field N |
| RDF | 6214 | Read data field |
| RIF | 6224 | Read instruction field |
| RIB | 6234 | Read interrupt buffer |
| RMF | 6244 | Restore memory field |
| CDI | 62n3 | Change to data field and instruction field n |

## PSEUDO-OPERATORS

The following is a list of the PAL III and MACRO-8 assembler pseudo-ops.

| Page Ref | PAL III | MACRO-8 |
|---|---|---|
| 23 | DECIMAL | DECIMAL |
| 24 | OCTAL | OCTAL |
| 24 | FIELD | FIELD |
| 26 | PAUSE | PAUSE |
| 23 | I | I |
| 23 | Z | Z |
| 27 | $ | $ |
| 27 | EXPUNGE | EXPUNGE |
| 27 | FIXTAB | FIXTAB |
| | | PAGE |
| 11-19 | = | = |
| 8→ | * | * |
| 28,29 | FIXMRI | DEFINE |
| | | DUBL |
| | | FLTG |
| | | TEXT |

Address arithmetic (PAL III), 11
Address assignments (PAL III), 17
Address field delimiter
  MACRO-8, 42
  PAL-11, 16
Ampersand (&) operator (MACRO-8), 38
AND group skip instructions
    (PAL III), 22
Arithmetic expressions (MACRO-8, 45
Arithmetic operators (PAL III), 14
ASCII code (PAL III), 29
ASCII-1 character set, B-1
Assembly listing (PAL III), 31
Assembler output
  MACRO-8, 69
  PAL-11, 29
Autoindexing (PAL III), 19

Binary punch routine
  MACRO-8, 65
  PAL III, 28
Binary tape (PAL III), 2
BIN (binary) loader, A-4
  loading, A-5, A-7

Character codes, B-1
Character set (PAL III), 2,3
Characters (MACRO-8), 37
Checksum (PAL III), 31
Coding practices (PAL III), 6
Combining memory reference instruc-
    tions (PAL III), 15
Comments (PAL III), 6
Compatibility - PAL III and
    MACRO-8, 61
Constants table (MACRO-8), 43
Current address indicator
    (PAL III), 17
Current location counter (PAL III),
    3, 8
  incrementation, 15, 17

DDT, 1
DECIMAL pseudo-op
  MACRO-8, 57
  PAL III, 3, 24
Defined symbols (MACRO-8), 60
DEFINE pseudo-op (MACRO-8), 53, 57
Delimiter for address field
    (PAL III), 16
Diagnostic messages, summary of
    PAL III, 34
Direct assignment statements
    (PAL III), 11
Dollar sign ($) (end of program)
    pseudo-op,
  MACRO-8, 57
  PAL III, 27
Double precision constants
    (MACRO-8), 50
Double precision floating point
    constants (MACRO-8), 51
Double precision integers
    (MACRO-8), 50
DUBL pseudo-op (MACRO-8), 57
Dummy arguments (MACRO-8), 53, 61
Duplicate tag (PAL III), 34

Editor, symbolic tape, 1
End of program (PAL III), 27
End of tape (PAL III), 26
Equal sign (PAL III), 12
Error diagnostics (MACRO-8), 66
Evaluation of a symbolic expression
    (MACRO-8), 38
Exclamation point (!) operator
    (MACRO-8), 38
Expressions
  MACRO-8, 37
  PAL-III, 14
    termination, 15
EXPUNGE pseudo-op
  MACRO-8, 57, 58
  PAL III, 27
Extended memory (PAL III), 24
Extended symbol tape (PAL III), 33

FIELD pseudo-op
  MACRO-8, 46, 57
  PAL III, 24
Field setting (PAL III), 24
FIXMRI pseudo-op (PAL III), 28, 29
FIXTAB pseudo-op
  MACRO-8, 57, 58
  PAL III, 27
Floating point (MACRO-8)
  constants, 51
  conversion accuracy, 53
  number format, 52
FLTG pseudo-op (MACRO-8), 52, 57
Format effectors (PAL III), 3
Form feed (PAL III), 3

Halts (MACRO-8), 72
Hardware configuration (PAL III), 2

I (indirect address) pseudo-op
  MACRO-8, 42, 57
  PAL III, 23
Illegal characters (PAL III), 34
Illegal reference (PAL III), 35
Inclusive OR (PAL III), 14
Incrementing current location
    counter (PAL III), 15, 17
Indirect addressing (PAL III), 18,23

Indirect address linkage (MACRO-8), 41
Input device (MACRO-8), 70
Input/output transfer micro-instructions (PAL III), 22
Instruction codes, C-1
Instructions (PAL III), 6, 19
Integers (MACRO-8), 49
Internal symbol representation
  MACRO-8, 58
  PAL III, 7

Labels (PAL III), 5
Leader code (PAL III), 29, 31
Limited symbol space, 62
Link generation (MACRO-8), 41
Literals (MACRO-8), 42
Loaders
  BIN (binary), A-4
  RIM (Read-In-Mode), A-1, A-2
Loading procedures, A-1
  PAL III, 32
Location counter (MACRO-8), 40
Logical operators (PAL III), 14

Macro definition (DEFINE) (MACRO-8), 53
  restrictions, 55
MACRO-8 programming, 37
Macros, user defined (MACRO-8), 53
Memory, extended (PAL III), 24
Memory reference instructions
  MACRO-8, 61
  PAL III, 15, 20
Microinstructions (PAL III), 21,
Minus (-) operator
  MACRO-8, 38
  PAL III, 14
Multiple assignments (PAL III), 13

Nested literals (MACRO-8), 44
Nonprinting characters (PAL III), 2
Normalized form (MACRO-8), 51
Numbers
  MACRO-8, 49
  PAL III, 3

OCTAL pseudo-op
  MACRO-8, 57
  PAL III, 3-24
ODT, 1
Operands (PAL III), 6
Operate microinstructions (PAL III),21
Operating instructions (MACRO-8), 70
Operating procedures
  MACRO-8, 69
  PAL III, 32
Operation code (PAL III), 20
Operators, arithmetic and logical
  MACRO-8, 38
  PAL III, 14
OR group skip instructions PAL III), 22
Origin setting (MACRO-8), 39
Origin (starting address) of source program (PAL III), 17
Output device (MACRO-8), 70, 72

PAGE pseudo-op (MACRO-8), 40, 57
Pages (MACRO-8), 39
PAL III programming, 1
Pass 1 (PAL III), 2, 31
  diagnostics, 34
Pass 2 (PAL III), 2, 31
  diagnostics, 35
Pass 3 (PAL III), 2, 31
  diagnostics, 36
Pass 3 output - assembly listing, (MACRO-8), 68
PAUSE (end of tape) pseudo-op
  MACRO-8, 57
  PAL III, 26
Period (.) character (PAL III), 17
Permanent symbols (MACRO-8), 60
Permanent symbol table (PAL III), 7
  alterations, 27, 28
Permanent symbol tables, C-1
Plus (+) operator
  MACRO-8, 38
  PAL III, 14
Processor deletion (MACRO-8), 71
Programming
  MACRO-8, 37
    techniques, 46
  PAL-III, 1
Programming hints, 62
Program preparation (PAL III), 29
Pseudo-operators, C-5
  MACRO-8, 56, 57, 59
  PAL III, 23
Punching new assembler tape (MACRO-8), 65
Pushdown list overflow (PAL III), 35

Radix control (PAL III), 3, 23
Redefinition (PAL III), 34
Referencing a macro (MACRO-8), 56
Registers, autoindex (PAL III), 19
RETURN key (PAL III), 4
RIM (Read-In-Mode) loader, A-1
  checking the loader, A-4
  programs, A-2
ROTL macro, advantages and disadvantages (MACRO-8), 55

Semicolon (;) as statement terminator (PAL III), 4
Single character text facility (MACRO-8), 47
Skip instructions (PAL III), 22
Source program preparation (PAL III), 5

Space operator
  MACRO-8, 38
  PAL III, 14
Special characters (MACRO-8), 59
Starting address (origin) of source
    program (PAL III), 17
Statements (PAL III), 5
Statement terminators (PAL III), 4
Switch option (MACRO-8), 65, 71
Symbol capacity (PAL III), 11
Symbol categories (MACRO-8), 59
Symbol definitions, addition or
    deletion (PAL III), 27
Symbolic addresses (PAL III), 8
Symbolic Editor (PAL III), 29
Symbolic expression (MACRO-8), 38
Symbolic instructions (PAL III), 10
Symbolic operands (PAL III), 10
Symbolic tape (PAL III), 29
Symbolic tape editor, 1
Symbols (PAL III), 7
Symbol space, limited, 62
Symbol table (MACRO-8), 57
  capacity, 57
  modification, 58
  revision, 65
Symbol table (PAL III), 7, 11, 35
Tabulation (PAL III), 3
Tape editor, symbolic, 1
Terminating an expression
    (PAL III), 15
Terminators of statements
    PAL III), 4
Text facility (MACRO-8), 47
TEXT pseudo-op (MACRO-8), 48, 57
Text strings (MACRO-8), 48
Trailer code (PAL III), 31
Two-pass assembler (PAL III), 1

Undefined address (PAL III), 35
Undefined symbols (MACRO-8), 60
User-defined macros (MACRO-8), 53
User-defined symbols
  MACRO-8, 60
  PAL III, 7
Using PAL III assembler, 33

Z (page zero reference) pseudo-op
  MACRO-8, 57
  PAL III, 23

# HOW TO OBTAIN SOFTWARE INFORMATION

## SOFTWARE NEWSLETTERS, MAILING LIST

The Software Communications Group, located at corporate headquarters in Maynard, publishes software newsletters for the various DIGITAL products. Newsletters are published monthly, and keep the user informed about customer software problems and solutions, new software products, documentation corrections, as well as programming notes and techniques.

There are two similar levels of service:

- The Software Dispatch
- The Digital Software News

The Software Dispatch is part of the Software Maintenance Service. This service applies to the following software products:

PDP-9/15
RSX-11D
DOS/BATCH
RSTS-E
DECsystem-10

A Digital Software News for the PDP-11 and a Digital Software News for the PDP-8/12 are available to any customer who has purchased PDP-11 or PDP-8/12 software.

A collection of existing problems and solutions for a given software system is published periodically. A customer receives this publication with his initial software kit with the delivery of his system. This collection would be either a Software Dispatch Review or Software Performance Summary depending on the system ordered.

A mailing list of users who receive software newsletters is also maintained by Software Communications. Users must sign-up for the newsletter they desire. This can be done by either completing the form supplied with the Review or Summary or by writing to:

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

## SOFTWARE PROBLEMS

Questions or problems relating to DIGITAL's software should be reported as follows:

### North and South American Submitters:

Upon completion of Software Performance Report (SPR) form remove last copy and send remainder to:

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

The acknowledgement copy will be returned along with a blank SPR form upon receipt. The acknowledgement will contain a DIGITAL assigned SPR number. The SPR number or the preprinted number should be referenced in any future correspondence. Additional SPR forms may be obtained from the above address.

### All International Submitters:

Upon completion of the SPR form, reserve the last copy and send the remainder to the SPR Center in the nearest DIGITAL office. SPR forms are also available from our SPR Centers.

## PROGRAMS AND MANUALS

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

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

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

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

USERS SOCIETY

DECUS, Digital Equipment Computers Users Society, maintains a user exchange center for user-written programs and technical application information. The Library contains approximately 1,900 programs for all DIGITAL computer lines. Executive routines, editors, debuggers, special functions, games, maintenance and various other classes of programs are available.

DECUS Program Library Catalogs are routinely updated and contain lists and abstracts of all programs according to computer line:

- PDP-8, FOCAL-8, BASIC-8, PDP-12
- PDP-7/9, 9, 15
- PDP-11, RSTS-11
- PDP-6/10, 10

Forms and information on acquiring and submitting programs to the DECUS Library may be obtained from the DECUS office.

In addition to the catalogs, DECUS also publishes the following:

| | |
|---|---|
| DECUSCOPE | -The Society's technical newsletter, published bi-monthly, aimed at facilitating the interchange of technical information among users of DIGITAL computers and at disseminating news items concerning the Society. Circulation reached 19,000 in May, 1974. |
| PROCEEDINGS OF THE DIGITAL EQUIPMENT USERS SOCIETY | -Contains technical papers presented at DECUS Symposia held twice a year in the United States, once a year in Europe, Australia, and Canada. |
| MINUTES OF THE DECsystem-10 SESSIONS | -A report of the DECsystem-10 sessions held at the two United States DECUS Symposia. |
| COPY-N-Mail | -A monthly mailed communique among DECsystem-10 users. |
| LUG/SIG | -Mailing of Local User Group (LUG) and Special Interest Group (SIG) communique, aimed at providing closer communication among users of a specific product or application. |

Further information on the DECUS Library, publications, and other DECUS activities is available from the DECUS offices listed below:

DECUS
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754

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

Please cut along this line.

# READER'S COMMENTS

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

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

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

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

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

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

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

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

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

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

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

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

Postage will be paid by:

digital

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