Digital Equipment Corporation
Maynard, Massachusetts

digital

8K FORTRAN SYSTEM

PROGRAMMER'S REFERENCE MANUAL
PDP-8 FAMILY

DEC-08-KFXB-D

PDP-8 FAMILY
8K FORTRAN SYSTEM
PROGRAMMER'S REFERENCE MANUAL

For additional copies order No. DEC-08-KFXB-D from Program Library,
Digital Equipment Corporation, Maynard, Massachusetts Price: $1.75

DIGITAL EQUIPMENT CORPORATION □ MAYNARD, MASSACHUSETTS

1st Printing June 1968
2nd Printing (Rev) August 1968
3rd Printing (Rev) April 1969

CONTENTS

CONTENTS (Cont)

CONTENTS (Cont)

CONTENTS (Cont)

PREFACE

This manual describes a version of FORTRAN II designed specifically for the PDP-8/I, 8/L, 8, 8/S, and 5 computers with at least 8K words of core memory and a high-speed reader and punch.

It is assumed that the reader is familiar with the basic concepts of the FORTRAN language. Several excellent texts are available for a more elementary approach to FORTRAN programming. "A Guide to FORTRAN Programming," by Daniel D. McCracken (published by John Wiley and Sons, Inc.) is recommended.

CHAPTER 1
8K FORTRAN

8K FORTRAN (acronym for FORmula TRANslation) is used interchangeably to designate both the 8K FORTRAN language and translator or compiler.

The 8K FORTRAN compiler is a computer program that enables the programmer to express his problem using English words and mathematical statements similar to the language of mathematics and acceptable to the computer. The compiler translates the programmer's source program into symbolic language, and then the symbolic version of the program is translated into relocatable binary code, that is, machine language, the language of the computer. The relocatable binary code, which is output on paper tape, is then loaded into the computer for solution of the problem.

The 8K FORTRAN system has the following features:

a. Subroutines
b. Two levels of subscripting
c. Function subprograms
d. Input/output supervisors
e. Relocatable output loaded by the Linking Loader
f. COMMON statements
g. I, F, E, A, X, and H format specification
h. Arithmetic and trigonometric library subroutines

The 8K FORTRAN system (hereafter referred to as FORTRAN) consists of a one-pass FORTRAN Compiler, SABR Assembler, Linking Loader, and a library of subprograms (see the appendices).

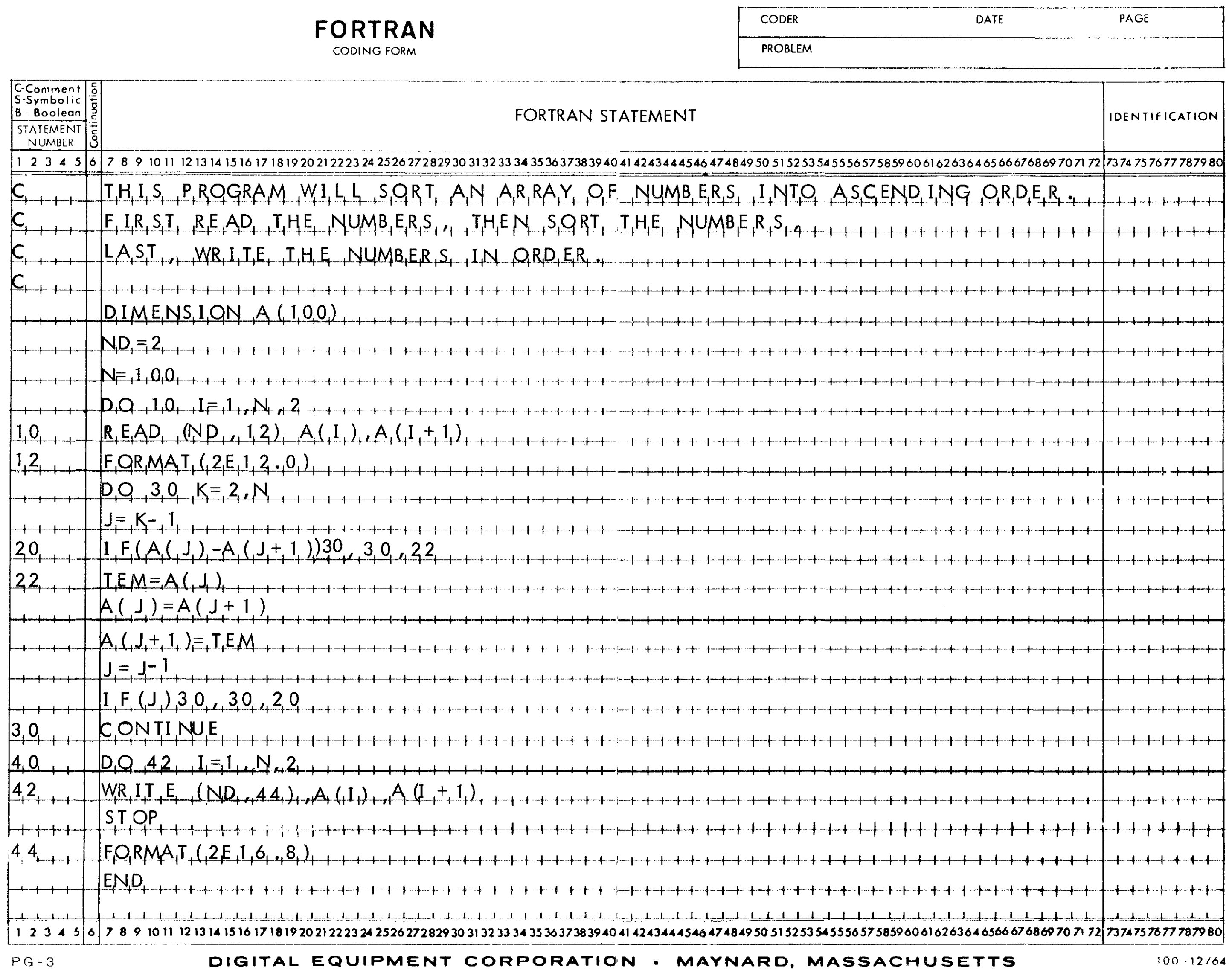
This FORTRAN system requires a PDP-8/I, 8/L, 8, 8/S, or 5 computer* with at least two fields of core memory, an ASR33 Teleprinter, and a high-speed paper tape reader and punch.

The appendices contain lists of the FORTRAN character set, statements, specifications, operating procedures for all phases of the system, error messages, and implementation notes.

1.1 LINE FORMAT

A line of data in FORTRAN is a string of 72 characters or less designated columns 1 through 72. Each line consists of three fields: statement number field, line continuation field, and statement field, as shown in Figure 1-1.

* The PDP-5 requires a PDP-8 extended memory control modification.



FORTRAN
CODING FORM

CODER | DATE | PAGE
PROBLEM

C-Comment / S-Symbolic / B - Boolean / STATEMENT NUMBER | Continuation | FORTRAN STATEMENT | IDENTIFICATION

1 2 3 4 5 | 6 | 7–72 | 73–80

```
C     THIS PROGRAM WILL SORT AN ARRAY OF NUMBERS INTO ASCENDING ORDER.
C     FIRST READ THE NUMBERS, THEN SORT THE NUMBERS.
C     LAST, WRITE THE NUMBERS IN ORDER.
C
      DIMENSION A(100)
      ND=2
      N=100
      DO 10 I=1,N,2
10    READ (ND,12) A(I),A(I+1)
12    FORMAT(2E12.0)
      DO 30 K=2,N
      J=K-1
20    IF(A(J)-A(J+1))30,30,22
22    TEM=A(J)
      A(J)=A(J+1)
      A(J+1)=TEM
      J=J-1
      IF(J)30,30,20
30    CONTINUE
40    DO 42 I=1,N,2
42    WRITE (ND,44) A(I),A(I+1)
      STOP
44    FORMAT(2E16.8)
      END
```

PG-3 DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS 100-12/64

Figure 1-1. Typical FORTRAN Coding Form

1.1.1 Statement Numbers

Each statement may have a positive integer as a label, which is used to reference that statement elsewhere in the program. A statement number consists of from one to four digits in columns 1 through 5. Statement numbers may be assigned nonsequentially; however, no two statements can have .he same number. Statement numbers must have a decimal value of 2047 or less.

1.1.2 Line Continuation Field

If a FORTRAN statement is so large that it cannot conveniently fit into one statement field, additional lines may be used to specify the complete statement. Any line which is not continued, or the first line of sequence of continued lines, must have a blank or zero in column 6. Continuation lines must have a character other than blank zero (must be a digit from 1 to 9 if a TAB is used in the label field) in column 6.

1.1.3 FORTRAN Statements

Any FORTRAN statement (listed below) may appear in the statement field (columns 7 through 72 or immediately following a tabulation (CTRL/TAB) character[1]). Each statement must begin on a separate line. Except for data within a Hollerith field (see Input/Output Statements), spaces are ignored and may be used freely for appearance purposes.

There are five types of FORTRAN statements:

a. Arithmetic Statements define calculations to be performed.

b. Control Statements govern the sequence of execution of statements within a program.

c. Input/Output Statements direct communication between the program and input/output devices.

d. Specification Statements describe the form and content of data within the program.

e. Subprogram Statements define the form and occurrence of subprograms and subroutines.

Each of the above statements is discussed in separate chapters of this manual.

1.2 COMMENTS

The letter C in column 1 of a line designates that line as a comment line. A comment has no effect upon the compilation process but it is listed on the printed output. There is no limit to the number of comment lines which may appear in a given program.

[1] A tabulation character is generated by typing CTRL/TAB, that is, holding down the CTRL key while depressing the TAB key.

1.3 CHARACTER SET*

The following characters are used in the FORTRAN language.

a. The alphabetic characters:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

b. The numeric characters:

0123456789

c. The special characters:

!	'
"	(
$	)
%	+
&	-
*	/
=	.
#	,
;	<
:	>
?	blank (space)

*Appendix A lists the octal and decimal representations of the FORTRAN character set.

CHAPTER 2
LANGUAGE ELEMENTS

The rules for defining constants and variables and for forming expressions are described below.

2.1 CONSTANTS

Constants are self-defining numeric values appearing in source statements. Two types of constants, integer and real, are permitted in a FORTRAN source program.

2.1.1 Integer Constants

Integer (fixed-point) constants are represented by a digit string of from one to four decimal digits, written with an optional sign and without a decimal point. An integer constant must fall within the range -2047 to +2047.

Examples:

47	
+47	(+ sign is optional)
-2	
0434	(leading zeros are ignored)
-0	(same as zero)

2.1.2 Real Constants

Real constants are represented by a digit string, an explicit decimal point, an optional sign, and possibly an integer exponent to denote a power of ten (7.2×10^3 is written 7.2E+03). A real constant may consist of any number of digits but only the leftmost eight digits appear in the compiled program. Real constants must fall within the range $.14 \times 10^{-38}$ to 1.7×10^{38}.

Examples:

+4.50	(+ is optional)
4.50	
-23.09	
-3.0E14	(same as -3.0×10^{14})

2.2 VARIABLES

A variable is a quantity whose value may change during execution of a program. Variables are specified by name and type. The name of a variable consists of one or more alphanumeric characters, the first of which must be alphabetic. Only the first five characters are interpreted as defining the variable name, the rest are ignored.

The type of variable (integer or real) is determined by the first letter of the variable name. A first letter of I, J, K, L, M, or N indicates an integer variable, and any other first letter indicates a real variable. Variables of either type may be either scalar or array variables. A variable is an array variable if it first appears in a DIMENSION statement.

2.2.1 Integer Variables

Integer variables undergo arithmetic calculations with automatic truncation of any fractional part. For example, if the current value of K is 5 and the current value of J is 9, J/K would yield 1 as a result.

Integer variables may be converted to real variables by the function FLOAT (see Section 2.3.2) or by an arithmetic statement. Integer variables must fall within the range -2048 to +2047.

2.2.2 Real Variables

A variable is a real variable when its name begins with any character other than I, J, K, L, M, or N. Real variables may be converted to integer variables by the function IFIX (see Section 2.3.2) or by an arithmetic statement. Real variables undergo no truncation in arithmetic calculations.

2.2.3 Scalar Variables

A scalar variable, which may be either integer or real, represents a single quantity.

Examples:

```
LM
A
G2
TOTAL
J
```

2.2.4 Array Variables

An array variable represents a single element of a one- or two-dimensional array of quantities. The variable is denoted by the array name followed by a subscript list enclosed in parentheses. The subscript list may be any integer expression or two integer expressions separated by a comma. The expressions may be arithmetic combinations of integer variables and integer constants. Each expression represents a subscript, and the values of the expressions determine the referenced array element. For example, the row vector A_i would be represented by the subscripted variable A(I), and the element in the second column of the first row of the matrix A, would be represented by A (1,2).

Examples:

```
Y(1)
PORT (K)
A (3*K+2, I)
```

The arrays above (Y, PORT, and A) would have to appear in a DIMENSION statement prior to their first appearance in an executable statement. The DIMENSION statement specifies the number of elements in the array.

Arrays are stored in increasing storage locations with the first subscript varying most rapidly and the last subscript varying least rapidly (see Appendix D). For example, the two-dimensional array B (J,K) is stored in the following order:

B(1,1),B(2,1),...,B(J,1),B(1,2),B(2,2),...,B(J,2),...,B(J,K)

2.3 EXPRESSIONS

An expression is a sequence of constants, variables, and function references separated by numeric operators and parentheses in accordance with mathematical convention and the rules given below.

Without parentheses, algebraic operations are performed in the following descending order:

**	exponentiation
-	unary negation
* and /	multiplication and division
+ and -	addition and subtraction
=	equals or replacement sign

Parentheses are used to change the order of precedence. An operation enclosed in parentheses is performed before its result is used in other operations. In the case of operations of equal precedence, the calculations are performed from left to right; this is also true for exponentiation.

Integers and real numbers may be raised to either integer or real powers. An expression of the form

A**B

means A^B and is real unless both A and B are integers. Exponential (e^x) and natural logarithmic ($\log_e(x)$) functions are supplied as subprograms (see Appendix E).

Excluding ** (exponentiation), no two numeric operators may appear in sequence unless the second is a unary plus or minus.

The mode (or type) of an expression may be either integer or real and is determined by its constituents. Variable modes may not be mixed in an expression with the following exceptions:

a. A real variable may be raised to an integer power.

A**2

b. Mode may be altered by using the functions IFIX and FLOAT.

A*FLOAT(I)

Any numeric expression may be enclosed in parentheses and be considered a basic element.

Example:

```
IFIX(X+Y)/2
(ZETA)
(COS(SIN(PI*EM)+X) )
```

A numeric expression may consist of a single element (constant, variable, or function call).

Example:

```
2.71828
Z(N)
TAN(THETA)
```

Compound numeric expressions may be formed using numeric operators to combine basic elements.

Example:

```
X+3.
TOTAL/A
TAN(PI*EM)
```

Alphabetic expressions preceded by a + or a – sign are also numeric expressions.

Example:

```
+X
-(ALPHA*BETA)
-SQRT(-GAMMA)
```

As an example of a typical numeric expression using numeric operators and a function call, the expression for the largest root of the general quadratic equation

$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

would be coded as

```
(-B+SQRT(B**2-4.*A*C))/(2.*A)
```

2.3.1 Function Calls

In addition to the basic numeric operators, function calls are provided to facilitate the evaluation of functions such as sine, cosine, and square root. A function is a subprogram which acts upon one or more quantities (arguments) to produce a single quantity called the function value. A function call may be used in place of a variable name in an arithmetic expression.

Function calls are denoted by the identifier which names the function (e.g., SIN, COS, etc.), followed by an argument list enclosed in parentheses as shown below.

identifier (argument, argument,...,argument)

At least one argument, which may be an expression or an array identifier, must be present (see Section 7.2). A function call is evaluated before the expression in which it is contained.

2.3.2 Library Subprograms

The standard FORTRAN library includes built-in functions, FUNCTION subprograms, and SUBROUTINE subprograms (see Chapter 7). Built-in functions are open subroutines, that is, they are incorporated into the object program each time they are referred to by the source program. FUNCTION and SUBROUTINE subprograms are closed subroutines; their names derive from the types of subprogram statements used to define them.

Table 2-1
Function Library

Name	Call	Definition	Argument
Absolute Value	ABS or IABS	$\lvert x \rvert$ $\lvert x \rvert$	Real Integer
Float	FLOAT	Conversion from integer to real	Integer
Fix	IFIX	Conversion from real to integer	Real
Remainder	IREM	Remainder of last integer divide*	Integer
Exponential	EXP	e^x	Real
Switch Register	IRDSW	Read console switch reg.	Integer
Natural Logarithm	ALOG	$\log_e(x)$	Real
Trigonometric Sine**	SIN	sine(x)	Real
Trigonometric Cosine**	COS	cos(x)	Real
Tangent**	TAN	tan(x)	Real
Square Root	SQRT	$(x)^{1/2}$	Real
Arctangent**	ATAN	arctan(x)	Real

* If IREM is called as IREM(I/J), the remainder of I/J will be returned. If the argument of IREM does not contain a division, the remainder of the last integer division will be returned.

** Trigonometric functions use radians rather than degrees.

The IRDSW function call (Switch Register) takes the decimal equivalence of the octal integer in the switch register as its result. For example, if the contents of the switch register is 1234 (668 in decimal) when the statement

N=IRDSW(0)

is executed, the switch register is read and its contents becomes the value of N, i.e.,

N=668

The switch register can be set in two ways:

1. Before executing the FORTRAN program, i.e., after pressing LOAD ADD and before pressing START.

2. During execution of the FORTRAN program, i.e., using the PAUSE statement.

CHAPTER 3
ARITHMETIC STATEMENT

One of the key features of FORTRAN is the ease with which arithmetic computations can be coded. Computations to be performed by FORTRAN are indicated by arithmetic statements, which have the general form

```
v=e
```

where v is a variable name (subscripted or nonsubscripted), e is an expression, and = is a replacement operator. The arithmetic statement causes the FORTRAN object program to evaluate the expression e and assign the resultant value to the variable v. Note that = signifies replacement, not equality. Thus, expressions of the form

```
A=A+B
```

and

```
A=A*B
```

are quite meaningful and indicate that the value of the variable A is to be changed.

Examples:

```
Y = 1.1*Y
P =X**2+3.*X+2.0
X(N) = EN*ZETA* (ALPHA+EM/PI)
```

The expression value is made to agree in type with the assignment variable before replacement occurs. For example, in the statement

```
META=W* (ABETA+E)
```

if META is an integer and the expression is real, the expression value is truncated to an integer before assignment to META.

CHAPTER 4
CONTROL STATEMENTS

FORTRAN compiled programs normally execute statements sequentially in the order in which they were presented to the compiler. However, the following control statements are available to alter normal sequence of statement execution: GO TO, IF, DO, PAUSE, STOP, and END.

4.1 GO TO STATEMENT

The GO TO statement has two forms: unconditional and computed.

4.1.1 Unconditional

Unconditional GO TO statements are of the form:

GO TO n

where n is the number of an executable statement. Control is transferred to the statement numbered n.

4.1.2 Computed

Computed GO TO statements have the form:

GO TO $(n_1, n_2, \ldots, n_k)$, J

where $n_1, n_2, \ldots, n_k$ are statement numbers and J is a nonsubscripted integer variable. This statement transfers control to the statement numbered $n_1, n_2, \ldots, n_k$ if J has the value 1, 2,..., k, respectively. If J is zero or if it exceeds the size of the list, execution will proceed to the next executable statement. For example, in the statement

GO TO (20,10,5),K

the variable K acts as a switch, causing a transfer to statement 20 if K=1, to statement 10 if K=2, or to statement 5 if K=3.

4.2 IF STATEMENT

Numerical IF statements are of the form:

IF (expression) n_1, n_2, n_3

where n_1, n_2, n_3 are statement numbers. This statement transfers control to the statement numbered n_1, n_2, n_3 if the value of the numeric expression is less than, equal to, or greater than zero, respectively. The expression may be simple.

Examples:

```
IF (ETA) 4,7,12
IF (KAPPA-L(10))20,14,14
```

4.3 DO STATEMENT

The DO statement simplifies the coding of iterative procedures. DO statements are of the form:

$$\text{DO n i}=m_1,m_2,m_3$$

where n is a statement number, i is a nonsubscripted integer variable, and m_1,m_2,m_3 are integer constants or nonsubscripted integer variables. If m_3 is not specified, it is understood to be 1.

The DO statement causes the statements which follow, up to and including the statement numbered n, to be executed repeatedly. This group of statements is called the range of the DO statement. In the example above, the integer variable i is called the index, the values of m_1,m_2,m_3 are, respectively, the initial, limit, and increment values of the index.

Examples:

```
DO 10 I=1,5,2
DO 10 I=J,K,5
DO 10 L=I,J,K
```

The index is incremented and tested before the range of the DO is executed. If the limit value is less than the initial value, the range of the DO will not be executed.

After the last execution of the range, control passes to the statement immediately following the range. This exit from the range is called the normal exit. Exit may also be accomplished by a transfer from within the range.

The range of a DO statement may include other DO statements, provided the range of each contained DO statement is entirely within the range of the containing DO statement. That is, the ranges of two DO statements must intersect completely or not at all. A transfer into the range of a DO statement from outside the range is not allowed.

Within the range of a DO statement, the index is available for use as an ordinary variable. After a transfer from within the range, the index retains its current value and is available for use as a variable[1]. The values of the initial, limit, and increment variables for the index and the index of the DO loop may not be altered within the range of the DO statement.

The last statement of a DO loop must be executable, and must not be an IF, GO TO, or DO statement.

[1] The index of a DO loop should not be used as a variable after a normal exit from that DO loop until it has been redefined.

4.4 CONTINUE STATEMENT

This is a dummy statement, used primarily as a target for transfers, particularly as the last statement in the range of a DO statement. For example, in the sequence

```
      DO 7 K=INIT,LIMIT
         .
         .
         .
      IF (X(K))22,13,7
         .
         .
         .
    7 CONTINUE
```

a positive value of X(K) begins another execution of the range. The CONTINUE provides a target address for the IF statement and ends the range of the DO statement.

4.5 PAUSE STATEMENT

The PAUSE statement enables the program to incorporate operator activity into the sequence of automatic events. The PAUSE statement assumes one of two forms:

```
      PAUSE
      PAUSE n
```

where n is an unsigned decimal number.

Execution of the PAUSE statement causes the octal equivalent of the decimal number n, to be displayed in the accumulator on the user's console. Program execution may be resumed (at the next executable statement) by depressing the CONTinue key on the console.

4.6 STOP STATEMENT

The STOP statement has the form:

```
      STOP
```

The STOP statement terminates the program.

4.7 END STATEMENT

The END statement has the form:

```
      END
```

The END statement informs the compiler to terminate compilation. The END statement must be the last statement of the program.

CHAPTER 5
INPUT/OUTPUT STATEMENTS

Input/output (I/O) statements are used to control the transfer of data between computer memory and peripheral devices and to specify the format of the output data. I/O statements may be divided into two categories.

a. A nonexecutable statement that enables conversion between internal data within core memory and external data: FORMAT

b. Data transmission statements which specify transmission of data between computer memory and I/O devices: READ and WRITE.

5.1 NONEXECUTABLE STATEMENT

The nonexecutable statement FORMAT enables the user to specify the form and arrangement of data on the selected external device.

5.1.1 FORMAT Statement

Nonexecutable FORMAT statements may be used with any appropriate input/output device. FORMAT statements are of the form:

$$\text{n FORMAT } (S_1, S_2, \ldots, S_n / S_1^1, S_2^1, \ldots, S_n^1 / \ldots)$$

where n is a statement number and each S is a data field specification.

FORMAT statements may be placed anywhere in the source program. Unless the FORMAT statement contains only alphanumeric data for direct input/output transmission, it will be used in conjunction with the list of a data transmission statement.

Unit records must be one of the following:

a. A paper tape record preceded by and followed by a carriage return/line feed.

b. A printed line with a maximum of 72 characters for a Teletype keyboard.

During transmission of data, the object program scans the designated FORMAT statement and if a specification for a numeric field is present (see Data Transmission Statements) and the data transmission statement contains items remaining to be transmitted, transmission takes place according to the specification. This process ceases and execution of the data transmission statement is terminated as soon as all specified items have been transmitted; thus, the FORMAT statement may contain specifications for more items than are specified by the data transmission statement. The FORMAT statement may also contain specifications for fewer items than are specified by the data transmission statement, in which case, format control will revert to the rightmost left parenthesis in the FORMAT statement (see Section 5.1.1.7).

Both numeric and alphanumeric field specifications may appear in a FORMAT statement. The FORMAT statement also provides for handling multiple record formats, skipping characters, space insertion, and repetition. If an input list requires more characters than the input device supplies for a given unit record, blanks are inserted.

5.1.1.1 Numeric Fields - Numeric field specification codes and the corresponding internal and external forms of the numbers are listed in the following table.

Table 5-1
Numeric Field Codes

Conversion Code	Internal Form	External Form
E	Binary floating point	Decimal floating point with E exponents: .324E+10
F	Binary floating point	Decimal floating point with no exponent: 283.75
I	Binary integer	Decimal integer: 79

Conversions are specified by the form:

```
rEw.d
rFw.d
rIw
```

where r is a repetition count, E, F, and I designate the conversion type, w is an integer specifying the field width, and d is an integer specifying the number of decimal places to the right of the decimal point. For E and F input, the position of the decimal point in the external field takes precedence over the value of d.

Example:

```
FORMAT (I5,F10.2,E16.8)
```

could be used to output the line

```
bbb32bbbb-17.60bbb.59625476E+03
```

on the output listing. (The letter b throughout this manual indicates the presence of a space.)

The field width should always be large enough to include the decimal point, sign, and exponent. In all numeric field conversions, if the field width is not large enough to accommodate the converted number, the excess digits on the left are lost; if the number is less than the field width, the number is right-adjusted in the field.

5.1.1.2 Alphanumeric Fields - Alphanumeric data can be transmitted in a manner similar to numeric data by use of the form

rAw

where A is the control character and w is the number of characters in the field. Alphanumeric characters are transmitted as the value of a variable in an input-output list; the variable may be either integer or real.

Although w may have any value, the number of characters transmitted is limited by the maximum number of characters which can be stored in the space allotted for the variable. This maximum depends upon the variable type; for a real variable the maximum is six characters, for an integer variable the maximum is two characters. If w exceeds the maximum, the leftmost characters are lost on input and replaced with blanks on output. If, on input, w is less than the maximum, blanks are filled in to the right of the given characters until the maximum is reached. If, on output, w is less than the maximum, the leftmost w characters are transmitted to the external device.

5.1.1.3 Hollerith Conversion - Alphanumeric data may be transmitted directly from the FORMAT statement by using Hollerith conversion (H). H-conversion format is referenced by WRITE statements only.

In H-conversion, the alphanumeric string is specified by the form

$$nH\ h_1,h_2,\ldots,h_n$$

where H is the control character and n is the number of characters in the string, including blanks. For example, the statement below can be used to print PROGRAM COMPLETE on the output listing.

FORMAT (17HbPROGRAMbCOMPLETE)

A Hollerith string may consist of any characters capable of representation in the processor. The space character is a valid and significant character in a Hollerith string. See also Section G.1.4 for an alternate method of outputting alphanumeric data.

5.1.1.4 Mixed Fields - An alphanumeric format field may be placed among other fields of the format. For example, the statement

FORMAT (I5,7HbFORCE=F10.5)

can be used to output the line:

bbb22bFORCE=bb17.68901

The separating comma may be omitted after an alphanumeric format field, as shown above.

5.1.1.5 Repetition of Fields - Repetition of a field specification may be specified by preceding the control character E,F, or I by an unsigned integer giving the number of repetitions desired.
For example:

FORMAT (2E12.4,3I5)

is equivalent to

FORMAT (E12.4,E12.4,I5,I5,I5)

5.1.1.6 Repetition of Groups - A group of field specifications may be repeated by enclosing the group in parentheses and preceding the whole with the repetition number.
For example:

FORMAT (2I8,2(E15.5,2F8.3))

is equivalent to

FORMAT (2I8,E15.5,2F8.3,E15.5,2F8.3)

5.1.1.7 Multiple Record Formats - To handle a group of output records where different records have different field specifications, a slash is used to indicate a new record. For example, the statement

FORMAT (3I8/I5,2F8.4)

is equivalent to

FORMAT (3I8)

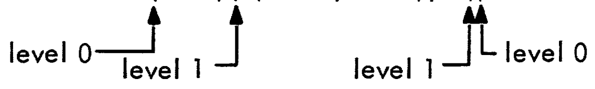
for the first record and

FORMAT (I5,2F8.4)

for the second record.

The separating comma may be omitted when a slash is used. When n slashes appear at the end or beginning of a format, n blank records may be written on output or ignored on input.
When n slashes appear in the middle of a format, n-1 blank records are written or n-1 records skipped. Both the slash and the closing parentheses at the end of the format indicate the termination of a record. If the list of an input/output statement dictates that transmission of data is to continue after the closing parenthesis of the format is reached, the format is repeated from the last open parenthesis of level one or zero. Thus, the statement:

FORMAT (F7.2,(2(E15.5,E15.4), I7))



causes the format:

F7.2,2(E15.5,E15.4), I7

to be used on the first record, and the format:

2(E15.5,E15.4),I7

to be used on succeeding records.

As a further example, consider the statement:

FORMAT (F7.2/(2(E15.5,E15.4),I7))

The first record has the format:

F7.2

and successive records have the format:

2(E15.5,E15.4),I7

5.1.1.8 Blank or Skip Fields - Blanks may be introduced into an output record or characters skipped on an input record by use of the specification nX. The control character is X; n is the number of blanks or characters skipped and must be greater than zero. For example, the statement:

FORMAT (5HbSTEPI5,10X2HY=F7.3)

may be used to output the line:

bSTEPbbb28bbbbbbbbbbY=bb3.872

5.2 DATA TRANSMISSION STATEMENTS

There are two data transmission statements, READ and WRITE. Data transmission statements accomplish input/output transfer of data that may be listed in a FORMAT statement. The data transmission statement contains a list of the quantities to be transmitted. The data appears on the external device in the form of records.

a. Input/Output Lists[1] - The list of an input/output statement specifies the order of transmission of the variable values. During input, the new values of listed variables may be used in subscript or control expressions for variables appearing later in the list.
For example,

READ(2,1000)L,A(L),B(L+1)

reads a new value of L and uses this value in the subscripts of A and B. Where 2 is the device designation code, and 1000 is a FORMAT statement number.

b. Input/Output Records - All information appearing on input is grouped into records. On output to the printer a record is one line. The amount of information contained in each ASCII record is specified by the FORMAT reference and the input/output list.

[1] The implied DO in input/output lists is not implemented.

Each execution of an input or output statement initiates the transmission of a new data record. Thus, the statement

```
READ(1,100)FIRST,SECOND,THIRD
```

is not necessarily equivalent to the statements where 100 is a FORMAT reference

```
READ(1,100)FIRST
READ(1,100)SECOND
READ(1,100)THIRD
```

since, in the second case, at least three separate records are required, whereas, the single statement

```
READ (d,f) FIRST,SECOND,THIRD
```

may require one, two, three, or more records depending upon FORMAT f.

If an input/output statement requests less than a full record of information, the unrequested part of the record is lost and cannot be recovered by another input/output statement without repositioning the record.

If an input/output list requires more than one ASCII record of information, successive records are read.

5.2.1 READ Statement

The READ statement specifies transfer of information from a selected input device to internal memory, corresponding to a list of named variables, arrays or array elements. The READ statement assumes the following form:

```
READ (d,f) list
```

where d is a device designation which may be an integer constant or an integer variable, f is a format reference, and list is a list of variables.

The first form of the READ statement causes ASCII information to be read from the device designated and stored in memory as values of the variables in the list. The data is converted to internal form as specified by the referenced FORMAT statement.

For example:

```
READ (1,15) ETA, PI
```

5.2.2 WRITE Statement

The WRITE statement is used to transmit information from the computer to a specified output device. The WRITE statement assumes one of the following forms:

```
WRITE (d,f) list
WRITE (d,f)
```

where d is a device designation (integer constant or integer variable) f is a format reference, and list is a list of variables.

The first form of the WRITE statement causes the values of the variables in the list to be read from memory and written on the device designated in ASCII form. The data is converted to external form as specified by the designated FORMAT statement.

The second form of the WRITE statement causes information to be read directly from the specified format and written on the device designated in ASCII form.

5.2.3 Device Designations

The I/O device designations are used in the READ and WRITE statements. The device codes are:

Device Code	Designating
1	Teletype and low-speed reader and punch
2	High-speed reader and punch

For additional I/O information, see SABR Manual, DEC-08-ARXA-D.

CHAPTER 6
SPECIFICATION STATEMENTS

Specification statements allocate storage and furnish information about variables and constants to the compiler. The specification statements are DIMENSION, COMMON, and EQUIVALENCE, and when used, must appear in the program prior to any executable statement.

6.1 COMMON STATEMENT

The COMMON statement causes specified variables or arrays to be stored in an area available to other programs. By means of COMMON statements, the data of a main program and/or the data of its subprograms may share a common storage area. Variables in COMMON statements are assigned to locations in ascending order in field 1 beginning at location 200 (see Appendix D). The COMMON statement has the general form:

COMMON $V_1 V_2, \ldots, V_n$

6.2 DIMENSION STATEMENT

The DIMENSION statement is used to declare identifiers to be array identifiers and to specify the number and bounds of the array subscripts. The information supplied in a DIMENSION statement is required for the allocation of memory for arrays. Any number of arrays may be declared in a single DIMENSION statement. The DIMENSION statement has the form:

DIMENSION $S_1, S_2, \ldots, S_n$

where S is an array specification.

Examples:

```
DIMENSION A (100)
DIMENSION Y(10),PORT(25),A(10,10),J(32)
```

NOTE

When variables in COMMON storage are dimensioned, the COMMON statement must appear before the DIMENSION statement.

6.3 EQUIVALENCE STATEMENT

The EQUIVALENCE statement causes more than one variable within a given program to share the same storage location. The EQUIVALENCE statement has the form:

$$\text{EQUIVALENCE } (V_1, V_2, \ldots), (V_k, V_{k+1}, \ldots), \ldots$$

where the V's are variable names.

The inclusion of two or more references in a parenthetical list indicates that the quantities in the list are to share the same memory location.

For example,

```
EQUIVALENCE (RED, BLUE)
```

specifies that the variables RED and BLUE are stored in the same place. The subscripts of array variables must be integer constants.

Example:

```
EQUIVALENCE (X,A(3),Y2,1)), (BETA(2,2),ALPHA)
```

Identifiers may not appear in both EQUIVALENCE and COMMON statements.

CHAPTER 7
SUBPROGRAM STATEMENTS

7.1 GENERAL

External subprograms are defined separately from (i.e., external to) the programs that call them, and are complete programs which conform to all the rules of FORTRAN programs. They are compiled as closed subroutines, that is, they appear only once in core memory regardless of the number or times they are used. External subprograms are defined by means of the statements FUNCTION and SUBROUTINE.

7.1.1 Dummy Identifiers

Subprogram definition statements contain dummy identifiers, representing the arguments of the subprogram. They are used as ordinary identifiers within the subprogram and indicate the sort of arguments that may appear and how the arguments are used. The dummy identifiers are replaced by the actual arguments when the subprogram is executed.

7.2 FUNCTION SUBPROGRAMS

A function subprogram is a single-valued function that may be called by using its name as a function name in an arithmetic expression, such as FUNC(N), where FUNC is the name of the subprogram that evaluates the corresponding function of the argument N. A function subprogram begins with a FUNCTION statement and ends with an END statement. It returns control to the calling program by means of one or more RETURN statements.

7.2.1 FUNCTION Statement

The FUNCTION statement has the form:

FUNCTION identifier $(a_1, a_2, \ldots, a_n)$

This statement declares the program which follows to be a function subprogram. The identifier is the name of the function being defined. This identifier must appear as a scalar variable and be assigned a value during execution of the subprogram which is the function value.

Arguments appearing in the list enclosed in parentheses are dummy arguments representing the function argument. The arguments must agree in number, order, and type with the actual arguments used in the calling program. Function subprogram may have expressions and array names as arguments.

Dummy arguments may appear in the subprogram as scalar identifiers or array identifiers. A function must have at least one dummy argument. Dummy arguments representing array names must appear within the subprogram in a DIMENSION statement. Dimensions must be given as constants and should be smaller than or equal to the dimensions of the corresponding arrays in the calling program.

A function should not modify any arguments which appear in the FORTRAN arithmetic expression calling the function. The only FORTRAN statements not allowed in a function subprogram are SUBROUTINE and another FUNCTION statement.

7.2.2 Function Type

The type of function is determined by the first letter of the identifier used to name the function, in the same way as variable names.

7.3 SUBROUTINE SUBPROGRAMS

A subroutine subprogram may be multivalued and can be referred to only by a CALL statement. A subroutine subprogram begins with a SUBROUTINE statement and returns control to the calling program by means of one or more RETURN statements.

7.3.1 SUBROUTINE Statement

The SUBROUTINE statement has the form:

SUBROUTINE identifier $(a_1, a_2, \ldots, a_n)$

This statement declares the program which follows to be a subroutine subprogram. The first identifier is the subroutine name. The arguments in the list enclosed in parentheses are dummy arguments representing the arguments of the subprogram. The dummy arguments must agree in number, order, and type with the actual arguments used by the calling program.

Subroutine subprograms may have expressions and array names as arguments. The dummy arguments may appear as scalar or array identifiers.

Dummy identifiers which represent array names must be dimensioned within the subprogram by a DIMENSION statement. The dummy arguments must not appear in an EQUIVALENCE or COMMON statement in the subroutine subprogram.

A subroutine subprogram may use one or more of its dummy identifiers to represent results. The subprogram name is not used for the return of results. A subroutine subprogram need not have any argument at all.

Examples:

```
SUBROUTINE FACTOR (COEFF,N,ROOTS)
SUBROUTINE RESID U(NUM,N,DEN,M,RES)
SUBROUTINE SERIES
```

The only FORTRAN statements not allowed in a function subprogram are FUNCTION and another SUBROUTINE statement.

7.3.2 CALL Statement

The CALL statement assumes one of two forms:

```
CALL identifier
CALL identifier (argument, argument,...argument)
```

The CALL statement is used to transfer control to a subroutine subprogram. The identifier is the subroutine name.

The arguments may be expressions or array identifiers. Arguments may be of any type, but must agree in number, order, type, and array size with the corresponding arguments in the SUBROUTINE statement of the called subroutine. Unlike a function, a subroutine may produce more than one value and cannot be referred to as a basic element in an expression.

A subroutine may use one or more of its arguments to return results to the calling program. If no arguments at all are required, the first form is used.

Examples:

```
CALL EXIT
CALL TEST (VALUE,123,275)
```

The identifier used to name the subroutine is not assigned a type and has no relation to the types of the arguments. Arguments which are constants or formed as expressions must not be modified by the subroutine.

7.3.3 RETURN Statement

The RETURN statement has the form

```
RETURN
```

This statement returns control from a subprogram to the calling program. Each subprogram must contain at least one RETURN statement. Normally, the last statement executed in a subprogram is a RETURN statement. Any number of RETURN statements may appear in a subprogram. The RETURN statement may not be used in a main program.

CHAPTER 8
OPERATING INSTRUCTIONS

This chapter describes how to compile, assemble, and execute a FORTRAN program using the 8K FORTRAN Compiler, SABR Assembler, and Linking Loader. The PDP-8/I System User's Guide, DEC-08-NGCB-D, is frequently referenced for loading instructions.

Except when loading the Linking Loader (Section 8.5), the DF setting is ignored because all other system tapes have field settings coded on the tapes.

8.1 LOADING THE COMPILER

8.1.1 Loading Into Core Memory

a. Make sure the Binary Loader is in memory, say field 1.

b. Place the FORTRAN Compiler binary tape in the reader.

c. Set the console switches as follows: (Data field is ignored) instruction field = 1, Switch Register = 7777.

d. Press LOAD ADDress.

e. Depress Switch Register bit 0.

f. Press START

g. The FORTRAN Compiler has now been loaded into memory by the Binary Loader. Parts of the compiler will load into field 0 and field 1.

8.1.2 Loading on the Disk

a. Make sure the Disk Monitor is in memory. (Type CTRL/C[1] or START at 7600.)

b. When the Monitor responds with a dot, call the system loader by typing

```
.LOAD ↵        (the ↵ denotes typing the RETURN key)
```

c. Place the Compiler binary tape in the reader.

d. Answer the Loader command dialogue as follows:

```
* IN-R: ↵
*
*OPT-2
*ST = ↵
↑ <CTRL/P> ↑ <CTRL/P> ↑ <CTRL/P> ↑ <CTRL/P>
.
```

[1] CTRL/C and CTRL/P are typed by holding down the CTRL key while typing the C or P key. They do not echo (print) when typed, therefore, their presence are indicated by being enclosed in angle brackets.

After typing the second CTRL/P remove the tape from the reader and place it back in the reader for the second pass.

e. The FORTRAN Compiler has now been loaded into memory, parts into field 0 and field 1. It must now be saved on the system device as follows:

```
.SAVE FTC0 ! 0 - 7577; 5363↲
.SAVE FTC1 ! 200, 1000 - 1577, 2600, 6000 - 16377;↲
```

f. The Compiler has now been saved on the user's system device and may be called as follows:

```
.FTC1↲
.FTC0↲
```

The field 1 part must be called first.

8.2 OPERATING THE COMPILER

It is assumed that the programmer has written his main program and possibly one or more subprograms, and that these source programs have been punched on paper tape in ASCII format. Remember that each source tape must have an END statement at the end of the tape.

After the Compiler has been loaded into memory, it is used to translate each FORTRAN statement into one or more SABR assembler instructions. The Compiler output will be punched in two parts separated by approximately three feet of blank tape. The first part, (executable code) will be punched as the source tape is read. The second part, (variable storage and constants) will be punched after the entire source tape has been read.

If the Compiler has been saved on the Disk Monitor System, it will halt after it is loaded into memory. Be sure that the source tape has been placed in the reader and the punch has been turned ON, then simply press CONTinue to begin step(d).

It may be desirable to suppress all compiler output the first time a particular program is compiled, simply to check for errors. To do this it is necessary to load the Compiler and then deposit 3075 in location 0356 (field 0), prior to executing step (c) below.

a. Set the console switches as follows: Data field = 0, Instruction field = 1 Switch Register = 1000. (The Compiler may also be started at location 5364 in field 0.)

b. Place the FORTRAN program source tape in the reader, and press the punch ON.

c. Press LOAD ADDress and START.

d. As soon as the Compiler has typed out an identification number, it will begin compiling the user's program. The Compiler output will generally be several times the length of the FORTRAN source program.

e. If an error is discovered in the user's FORTRAN program, the Compiler will type the incorrect line, followed by an error message. Although compiler output will be suppressed, the rest of the user's program will be read, and additional error messages may be typed.

f. When the Compiler has finished punching both sections of tape it will halt. It may be restarted to compile additional programs by pressing CONTinue.

g. The FORTRAN Compiler may be restarted at any time by pressing STOP and going back to step (a).

8.3 LOADING THE SABR ASSEMBLER

8.3.1 Loading Into Core Memory

a. Make sure the Binary Loader is in memory, say in field n.

b. Set the console switches as follows: Instruction Field = n, Switch Register = 7777. (Data field is ignored)

c. Press LOAD ADDress.

d. Insert the SABR binary tape into the reader.

e. If using the high-speed reader depress Switch Register Bit 0.

f. Press START.

g. SABR will now be loaded into memory by the Binary Loader. Portions of SABR will load into Field 0 and Field 1.

8.3.2 Loading on the Disk

a. Make sure the Disk Monitor is in memory. (Type CTRL/C or START at 07600.)

b. When the Monitor responds with a dot, call the system Loader as follows:

```
.LOAD ↵
```

c. Insert the SABR binary tape in the reader.

d. Answer the loading command dialogue as follows:

```
*IN-R:↵  for high-speed reader or     *IN-T: ↵  for ASR reader
*
*OPT -2
*ST = ↵
↑ <CTRL/P>  ↑ <CTRL/P>  ↑ <CTRL/P>  ↑ <CTRL/P>
.
```

After typing the second CTRL/P remove the tape from the reader and place it back in the reader for the second pass.

e. SABR is now loaded into memory, partly in Field 0 and partly in Field 1. It may be saved on the user's system device by responding to the monitor's dot as follows:

```
. SAVE SABR! 0-7177; 200 ↵
. SAVE SAB1! 12000 - 12427;↵
.
```

f. SABR is now saved on the user's system device and may be called as follows:

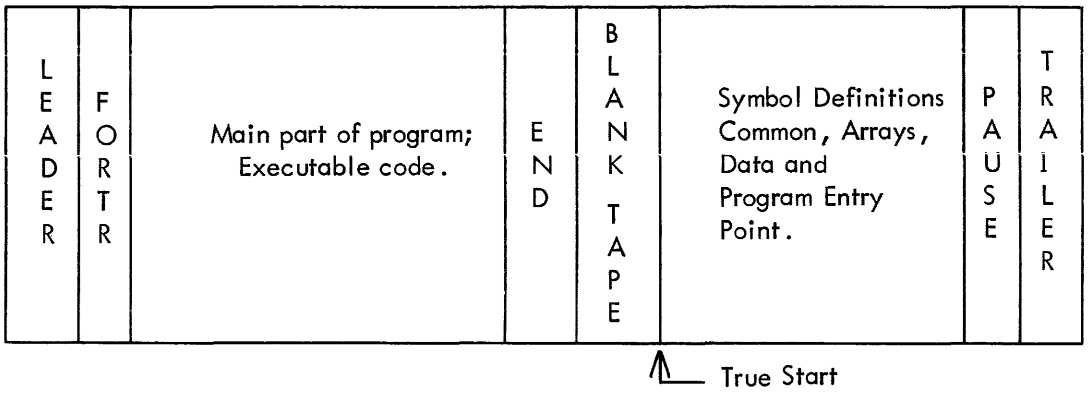
```
.SAB1 ↲
.SABR ↲
```

The Field 1 portion must be called first.

8.4 OPERATING THE SABR ASSEMBLER

In addition to being a stand-alone assembler, SABR also serves as the second pass of 8K FORTRAN compilation. For this purpose the use of SABR is slightly different from that described in the SABR manual. This difference in the operation of SABR is due only to the unusual format of the FORTRAN Compiler output.

The Compiler, in one pass, converts the user's FORTRAN source into a symbolic machine language program tape. SABR then converts the symbolic tape into relocatable binary. However, the symbolic tape produced by the Compiler is not a standard format SABR language tape. It is arranged as shown in the figure below.


LEADER
FORTR
Main part of program; Executable code.
END
BLANK TAPE
Symbol Definitions Common, Arrays, Data and Program Entry Point.
PAUSE
TRAILER
True Start

The tape is arranged this way because the data at the end of the tape cannot be inserted in the midst of the executable code, and some of it which should be at the beginning of the tape is not known until later. Thus the true start of the symbolic program is near the end of the symbolic tape preceded by a segment of leader/trailer code and followed by a PAUSE statement.

To assemble such a tape with SABR one of three methods must be followed. Actually, the general procedure is the same as that described in the SABR manual, but in particular details it differs. The differences are covered by the three methods explained below.

8.4.1 Method 1

The simplest method is to cut the symbolic tape into two parts. The cut should be made at the middle of the blank tape which separates the executable code from the symbol definitions. The latter section of the tape should then be marked "Section 1" and the former section (the executable code) should be marked "Section 2." Assembly then proceeds with the two part symbolic tape exactly as described below.

After SABR has been loaded into memory, it is used to assemble the Compiler output. In the first pass through SABR the relocatable binary version of the user's program is created and, at the end of this pass, the symbol table may be typed and/or punched. Pass 2 is the listing pass. The assembly is carried out as follows.

If SABR has been saved on the system I/O device as in Section 8.3, it will start automatically at step (c) below when called into memory. The source tape (first section) should be inserted in the reader before operation begins.

It may be desirable to suppress all assembler output the first time a particular program is assembled, simply to check for errors. To do this it is necessary to load SABR and then deposit 5370 in location 3165 (Field 0) before beginning step (a) below.

a. Set the console switches as follows: Data field = 0, Instruction field = 0, Switch Register = 0200.

b. Press LOAD ADDress and START.

c. SABR now types a sequence of two or three questions;

```
"HIGH SPEED READER?"
"HIGH SPEED PUNCH?"
"LISTING ON HIGH SPEED PUNCH?"
```

These questions must be answered with "Y" if the answer is "yes." Any other answer is assumed to be "no." The third question is typed only if the second is answered "Y". If the third is answered "Y," both the symbol table and the listing will be punched on the high-speed paper tape punch. Otherwise, they are typed on the teletypewriter. Incidentally, the user need not wait for the full question to be typed before responding.

d. As soon as SABR has echoed the user's response to the last question, the punch device and, if it is being used, the ASR reader should be turned on. If using the low-speed reader, the error message E indicates that the user has waited too long before turning the reader on. He will have to start over.

e. At this point, Pass 1 begins. SABR reads the source tape and punches the binary tape. After the binary tape has been completed SABR will type or punch the program symbol table.

f. If the source tape is in several sections (separate tapes with PAUSEs at the end of all except the last), SABR will halt at the end of each section. At this point the user should insert the next section in the reader and then press CONTinue.

g. At the end of Pass 1 SABR will halt.

h. If the user desires an assembly listing, he should now reposition the beginning of the source tape in the reader and press CONTinue.

If the listing is going to be punched on the high speed punch, the user may want to list the symbol table (at the end of the binary relocatable tape) before beginning Pass 2.

i. At the end of Pass 2 SABR will again halt. It may be restarted for assembling another program by pressing CONTinue.

j. SABR may be restarted at any time by pressing STOP, setting the Switch Register = 0200, pressing LOAD ADDress and START. However, Pass 1 must always be repeated.

8.4.2 Method 2

The user may avoid actually cutting the symbolic tape by cleverly manipulating the tape as if it were two parts as explained above. The tape should initially be inserted in the reader with the separator blank tape over the read-head. When SABR halts at the PAUSE statement at the physical end of the tape, the user should reposition the tape, putting the physical beginning of the tape in the reader. Then press CONTinue. The assembly pass will end at the separator blank tape code. The assembly listing can be produced in a similar manner, pressing CONTinue to start the Listing pass.

8.4.3 Method 3

The third method requires SABR to pass over the symbolic tape two times for each pass of the assembly. However, it allows the tape to be inserted at its physical beginning. It is based on the fact that a symbolic tape output by the FORTRAN Compiler has as its physical first line the special pseudo-op, FORTR. This pseudo-op has no effect except when a symbolic tape output by the Compiler is assembled using this third method.

The method is this:

a. Insert the symbolic tape in the reader at its physical beginning.

b. Start SABR as usual.

c. Sensing the FORTR statement as the first line, SABR ignores all further data until after it passes over the END statement. SABR then begins the actual assembly by processing the symbol definitions, etc., which are at the latter end of the tape.

d. Then SABR halts at the PAUSE statement which is at the physical end of the tape. At this time the user should reposition the symbolic tape in the reader at the physical beginning of the tape, and then press CONTinue. SABR will now assemble the executable code portion of the tape in the normal way.

e. If the user desires an assembly listing, he should proceed as in Method 2 after SABR finishes the assembly pass.

One further type of error may occur. This is an undefined symbol. Because SABR is a one pass assembler, this can not be determined until the end of the assembly pass, so the error diagnostic UNDF is given in the symbol table listing.

8.5 THE LINKING LOADER

Relocatable binary program tapes produced by SABR are loaded into memory by the 8K System Linking Loader. The Linking Loader is capable of loading and linking a user's program and subprograms in any fields of memory. It is even capable, in a special way, of loading programs over itself. The Linking Loader also has options which give storage maps and core availability.

Generally speaking, the Linking Loader is capable of loading any number of user and Library programs into any field of memory. These programs are loaded one after the other via the high-speed reader (or the ASR reader). The choice of which field to load each program into is a Switch Register option. Usually several programs may be loaded into each field. Because of the space reserved for the Linkage Routines, the available space in Field 0 is three pages smaller than in all other fields.

Any COMMON storage reserved by the programs being loaded is allocated in Field 1 from location 0200 upwards. The space reserved for COMMON is obviously subtracted from the available loading area in Field 1. The program reserving the largest amount of COMMON storage must be loaded first.

The Linking Loader uses the following special method to enable loading data over itself. When the Linking Loader encounters data which must be loaded over itself, it punches this data onto paper tape in RIM format. Then after the user has finished loading all his relocatable binary program tapes, all that is necessary is to load the RIM format tape using the RIM loader.

The Run-Time Linkage Routine necessary to execute SABR programs are automatically loaded into the required areas of every field by the Linking Loader as a part of its initialization. The user needs to know nothing more about these routines than the particular areas of core they occupy. (See Appendix D of the SABR manual.)

The 8K System Library subprograms, which may be used by any SABR program, are loaded in the same way as any other relocatable binary programs. Only those Library programs which the user's programs actually call need to be loaded. Refer to the SABR manual for additional information.

During the loading operation with the Linking Loader, two options are available to the user to obtain information about what has been loaded so far.

The Switch Register is used to select these options. Either option may be selected after any program has finished loading. (Warning: if the ASR punch is turned on, it must be turned off before selecting these options.) The Switch Register bits used are as follows.

BIT 0 = 1 selects the Core Availability option;
BIT 1 = 1 selects the Storage Map option.

The Core Availability option, when selected, causes the number of free pages of memory in every field of memory to be typed in a list on the teletype. For example, if the user has a 16K configuration a list like the following might be typed.

```
0002          (number of free pages in field 0)
0010          (number of free pages in field 1)
0030          (number of free pages in field 2)
0036          (number of free pages in field 3)
```

The number of pages initially available in field 0 is 0033 and in all other fields is 0036.

The Storage Map option, when selected, causes a list of all program Entry points to be typed along with the actual address at which they have been loaded. Entry points of programs which have been called but which have not been loaded are also listed along with a U flag for "undefined." Such flagged programs must be loaded before execution of the user's programs are possible. The core availability list is automatically appended to the storage map. A sample is shown below.

```
MAIN       10200
READ       01055
WRITE      01066
I OH       03031
SETERR     00000  U
ERROR      00000  U
T TYOUT    00000  U
H SOUT     00000  U
TTYIN      00000  U
H SIN      00000  U
F DV       04722
CLEAR      05247
I FAD      05131
FMP        04632
ISTO       05074
STO        04447
FLOT       05210
FAD        04010
DIV        00000  U
I REM      00000  U
F SB       04000
FLOAT      05046
FIX        04513
IFIX       04561
CHS        05231
0011
0033
```

8.6 LOADING THE LINKING LOADER

The Linking Loader must be loaded into the highest available field of memory.

a. Make sure the Binary Loader is in memory, say in field 1.

b. Set the console switches as follows: Data field = h, Instruction field = 1, Switch Register = 7777. Where h represents the number of the highest field in the user's configuration.

c. Press LOAD ADDress.

d. Place the binary paper tape of the Linking Loader in the reader.

e. If using a high-speed reader, depress Switch Register Bit 0.

f. Press START. The Linking Loader will now be loaded into memory.

8.7 OPERATING THE LINKING LOADER

The Linking Loader is used to load the user's relocatable programs and 8K system Library subprograms as outlined below.

The program or subprogram which uses the largest amount of COMMON storage should be loaded first. (The Library subprograms do not use COMMON.)

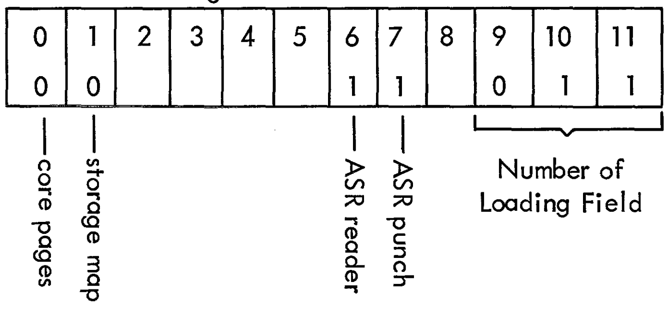
a. After the Linking Loader has been loaded into the highest memory field, h, the user should set the console switches as follows: Data Field = h, Instruction Field = h, Switch Register =0200.

b. Press LOAD ADDress.

c. Place the relocatable binary tape for the first program to be loaded in the reader. It should be positioned with leader code in the reader.

d. The Switch Register should be set to 0000. Then, if loading via the ASR reader is required, raise Switch Register Bit 6. If the user does not have high-speed punch, he should raise Switch Register Bit 7. Finally the user should set Switch Register Bits 9-11 to the number of the field into which he wishes to load the first program or subprogram.

Switch Register*



Example:

If the user wishes to load his first program into Field 3, and if he has no high-speed I/O device, then he should set the Switch Register to 0063 before the next step.

* All other Switch Register bits are irrelevant.

e. Press START.

f. The user's relocatable binary program will now be loaded. When loading is completed the Linking Loader will halt.

g. The user may now either load another program or select one of the options.

h. To load another program, insert the program relocatable binary tape in the reader, set Switch Register Bits 9-11 to the number of the field the program is to be loaded into, and then press CONTinue.

i. To select the Core Availability option, set Switch Register Bit 0 = 1 and press CONTinue.

j. To select the Storage Map option, set Switch Register Bit 1 = 1 and press CONTinue.

WARNING

If the ASR punch is turned on for possible RIM format data punching (as explained in Section 6.2), be sure to turn it off before selecting either of the options and to turn it back on after the typing of the option is completed.

k. The user may continue loading more programs as in step (h) after using either of the options.

Any time the Linking Loader halts the user may access memory directly via the DEPosit and EXAMine console switches. After this is done the Linking Loader may be restarted via the console switches at location 7200 (in the highest field, where the Linking Loader resides).

In general, all five parts of FORTRAN Library Tape I must be loaded before any FORTRAN program can be executed. The five parts of Library II may be loaded selectively as determined by the Storage Map option, and the following table:

8.7.1 Library Organization

Tape I	1)	"IOH"	contains	IOH, READ, WRITE
	2)	"FLOAT"	contains	FAD, FSB, FMP, FDV, STO, FLOT, FLOAT, FIX, IFIX, IFAD, ISTO, CHS, CLEAR
	3)	"INTEGER"	contains	IREM, ABS, IABS, DIV, MPY, IRDSW
	4)	"UTILITY"	contains	TTYIN, TTYOUT, HSIN, HSOUT, OPEN, CKIO
	5)	"ERROR"	contains	SETERR, CLRERR, ERROR
Tape II	1)	"SUBSC"	contains	SUBSC (Subscripting routine)
	2)	"POWERS"	contains	IIPOW, IFPOW, FIPOW, FFPOW, EXP, ALOG
	3)	"SQRT"	contains	SQRT
	4)	"TRIG"	contains	SIN, COS, TAN
	5)	"ATAN"	contains	ATAN

8.8 EXECUTING THE FORTRAN PROGRAM

Determine the starting address of your Main program by using the Linking Loader Storage Map option. The address will be typed in the form:

```
MAIN     dnnnn
```

a. Set Data field = d, Instruction field = d, Switch Register = nnnn.

b. Turn on paper tape punch and/or put data tape in reader as required.

c. Press LOAD ADDress, and START.

   Program execution will begin.

CHAPTER 9
DEMONSTRATION PROGRAM

The purpose of this program is to compute the factorials of the even integers from 1 through 34. The MAIN program calls the subprogram to perform the computation.

This demonstration program was run on a PDP-8/I with 8K words of core memory and a high-speed photoelectric reader and punch. The demonstration, from start to finish, required 15 minutes. Actual Teletype printout is used below.

```
L

C       FORTRAN DEMONSTRATION PROGRAM
        DIMENSION A(35)
        DO 10 N=2,34,2
        A(N)=FACT(N)
10      WRITE (1,60)N,A(N)
        STOP
60      FORMAT (I3,4H! = ,E12.7)
        END

P
```

Both source programs were written using the Symbolic Editor, listed on the Teletype for inclusion here, and punched on the high-speed punch.

```
L

C       FORTRAN FUNCTION TO COMPUTE FACTORIALS
        FUNCTION FACT(N)
        IF (N-34) 1,5,5
1       IF (N) 2,4,2
2       M=N-2
        FACT=N
        DO 3 K=1,M
        C=N-K
3       FACT=FACT*C
        RETURN
4       FACT=1.
        RETURN
5       WRITE (1,6) N
        FACT=0
        RETURN
6       FORMAT (I5,30H! EXCEEDS CAPACITY OF MACHINE.)
        END

P

PDP-8 FORTRAN DEC-08-A2B1-3
```

This is the system program tape identification.

Loaded the FORTRAN Compiler and compiled both source programs.

```
PDP-8 SABR DEC-Ø8-A2B2-1Ø
HIGH SPEED READER?  Y
HIGH SPEED PUNCH? Y
LISTING ON  HIGH SPEED PUNCH? N


CKIO    ØØØØEXT
FACT    ØØØØEXT
IOH     ØØØØEXT
ISTO    ØØØØEXT
MAIN    Ø352EXT
OPEN    ØØØØEXT
SUBSC   ØØØØEXT
WRITE   ØØØØEXT
[Ø      Ø512
\A      Ø2ØØ
\N      Ø351
\1Ø     Ø426
\6Ø     Ø5Ø1
↑A      Ø361
↑B      Ø473
↑C      Ø411
↑D      Ø45Ø
↑E      Ø463
↑F      Ø476
↑G      Ø512


HIGH SPEED READER?  Y
HIGH SPEED PUNCH? Y
LISTING ON  HIGH SPEED PUNCH? N


FACT    Ø215EXT
FAD     ØØØØEXT
FLOT    ØØØØEXT
FMP     ØØØØEXT
IOH     ØØØØEXT
OPEN    ØØØØEXT
STO     ØØØØEXT
WRITE   ØØØØEXT
[Ø      Ø473
\C      Ø2Ø5
\FACT   Ø2Ø1
\K      Ø2Ø4
\M      Ø2ØØ
\N      Ø471
\1      Ø251
\2      Ø261
\3      Ø331
\4      Ø354
\5      Ø4Ø6
\6      Ø445
]3      Ø21Ø
↑A      Ø3Ø5
↑B      Ø346
↑C      Ø422
↑D      Ø471
```

Loaded the SABR Assembler, responded to the initial dialogue and assembled both compiled programs.

```
DEC-Ø8-A2B3-5

MAIN    Ø1152
OPEN    1Ø325
SUBSC   11ØØØ
FACT    Ø1415
ISTO    Ø6Ø62
WRITE   Ø2Ø66
IOH     Ø3744
CKIO    1Ø321
FLOT    Ø62ØØ
STO     Ø5444
FAD     Ø5Ø1Ø
FMP     Ø5623
READ    Ø2Ø55
SETERR  1Ø4ØØ
ERROR   1Ø5Ø3
TTYOUT  1Ø227
HSOUT   1Ø255
TTYIN   1Ø2ØØ
HSIN    1Ø245
FDV     Ø5711
CLEAR   Ø6237
IFAD    Ø6117
DIV     Ø6443
IREM    Ø6616
FSB     Ø5ØØØ
FLOAT   Ø6Ø34
FIX     Ø551Ø
IFIX    Ø5556
CHS     Ø6221
ABS     Ø6636
IABS    Ø67ØØ
MPY     Ø64ØØ
IRDSW   Ø6723
EXIT    1Ø344
CLRERR  1Ø431
ØØØ3
ØØ32
```

Loaded the Linking Loader and the Library programs (all of the first tape and the first section of the second tape, and then set the switch register for the memory map).

Loaded the relocatable binary tapes and started the MAIN program at location 01152 (see memory map).

Received the expected results.

```
 2! = .2ØØØØØØE+Ø1
 4! = .24ØØØØØE+Ø2
 6! = .72ØØØØØE+Ø3
 8! = .4Ø32ØØØE+Ø5
1Ø! = .36288ØØE+Ø7
12! = .479ØØ16E+Ø9
14! = .8717829E+11
16! = .2Ø92279E+14
18! = .64Ø2374E+16
2Ø! = .24329Ø2E+19
22! = .11240Ø1E+22
24! = .62Ø4484E+24
26! = .4Ø32915E+27
28! = .3Ø48883E+3Ø
3Ø! = .2652529E+33
32! = .26313Ø8E+36
  34! EXCEEDS CAPACITY OF MACHINE.
34! = .ØØØØØØØE+ØØ
```

APPENDIX A
DECIMAL AND OCTAL REPRESENTATIONS OF THE CHARACTER SET

This version of FORTRAN uses six-bit characters. Other ASCII characters will be trimmed to six bits and translated to even parity, resulting in one of the characters in the table.

A*	B*	C*
Carriage Return	0	00
A	1	01
B	2	02
C	3	03
D	4	04
E	5	05
F	6	06
G	7	07
H	8	10
I	9	11
J	10	12
K	11	13
L	12	14
M	13	15
N	14	16
O	15	17
P	16	20
Q	17	21
R	18	22
S	19	23
T	20	24
U	21	25
V	22	26
W	23	27
X	24	30
Y	25	31
Z	26	32
[	27	33
\	28	34
]	29	35
↑	30	36
←	31	37
Space	32	40
!	33	41
"	34	42
#	35	43
$	36	44
%	37	45
&	38	46
'	39	47
(	40	50
)	41	51
*	42	52
+	43	53
,	44	54
-	45	55
.	46	56
/	47	57
0	48	60
1	49	61
2	50	62
3	51	63
4	52	64
5	53	65
6	54	66
7	55	67
8	56	70
9	57	71
:	58	72
;	59	73
<	60	74
=	61	75
>	62	76
?	63	77

A* = character
B* = decimal representation
C* = octal equivalent

APPENDIX B
STATEMENT SPECIFICATIONS

STATEMENT		FORM (R or P indicates a required or prohibited statement number, N indicates nonexecutable statement)	WHERE
COMMENT	P	"C" in column 1	columns 2 through 80 will be ignored.
CONTINUE		CONTINUE	control goes to next statement.
ARITHMETIC		v=e	variable name = expression.
GO TO		GO TO n	n is a statement number.
		GO TO $(n_1, \ldots, n_m)$, i	$1 \leq i \leq m$ and control goes to statement n_i. i is a nonsubscripted integer variable.
IF		IF (E) n_1, n_2, n_3	control goes to $\begin{Bmatrix} n_1 \\ n_2 \\ n_3 \end{Bmatrix}$ if expression E$\begin{Bmatrix} \leq \\ = \\ \geq \end{Bmatrix}$ 0.
DO		DO n $i=m_1, m_2, m_3$	repeated execution through statement n beginning with $i=m_1$, incrementing by m_3, while i is less than or equal to m_2. m's and i may not be subscripted.
		DO n $i=m_1, m_2$	m_3 assumed to be 1.
PAUSE		PAUSE	temporary halt, resumed by CONTinue key.
		PAUSE n	octal equivalent of the integer n displayed.
STOP		STOP	must be used to halt execution of a main program.
		STOP n	octal equivalent of the integer n displayed.
END	NP	END	an END statement at the end of a subprogram tells the compiler there is no more program.

STATEMENT		FORM	WHERE
READ WRITE		READ (d,f) L WRITE (d,f) L	d is device number, f is a FORMAT statement number and L is list of variable names separated by commas.
FORMAT	NR	FORMAT ($k_1,\ldots,k_n$)	k's are format specifications
COMMON*	NP	COMMON a,b,...,n	a,...,n are nonsubscripted variable names
DIMENSION	NP	DIMENSION a_1 (k_1),...,a_n (k_n)	a's are array names and k's are maximum subscripts.
FUNCTION	NP	FUNCTION name ($a_1,\ldots,a_n$)	a's are dummy arguments and function name must be defined as a variable containing the value of the function.
SUBROUTINE	NP	SUBROUTINE name ($a_1,\ldots,a_n$)	a's are dummy arguments and subroutine name may not appear elsewhere in the subroutine.
CALL		CALL name ($a_1,\ldots,a_n$)	a's are actual arguments of a subroutine and may be expressions.
RETURN		RETURN	for subroutines, control returned to statement following CALL. For functions, evaluation of expression in calling program is resumed using value of the function.
EQUIVALENCE	NP	EQUIVALENCE ($V_1,\ldots,V_n$),..., ($V_m,\ldots,V_p$)	V's are variables or subscripted array names.

APPENDIX C
FORMAT SPECIFICATIONS

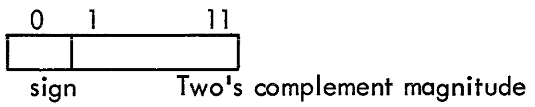
KIND	FORM	WHERE
Integer	rIw	r is the repetition count; w is total field width in characters.
Floating Point (Decimal)	rFw.d	r is the repetition count, w is field width including sign and decimal point, and d is number of characters to right of decimal.
Exponential	rEw.d	r is the repetition count, w is field width including sign, decimal point, and d is the number of characters in exponent.
Alphanumeric	rAw	r is the repetition count, w is field width.
H (Hollerith or Literal)	nHcharacters 'characters'	n is total number of characters following H. Parentheses in each format statement must balance. Characters enclosed within single quotes (SHIFT/7) are also printed.
Parentheses	n (specification)	format specification in parentheses is repeated n times.
Carriage Control	/	indicates beginning of a new data record.

APPENDIX D
STORAGE ALLOCATION

D.1 REPRESENTATION OF CONSTANTS AND VARIABLES

D.1.1 Integers

Integers are each allocated one machine word. They are represented in two's complement binary.



Positive numbers in two's complement binary are represented as straight binary with the first bit zero.

$\boxed{0}\boxed{11111111111}$

$3777_8 = +2047_{10}$, the largest positive integer.

Negative numbers are represented by replacing each 0 bit with a 1 and each 1 bit with a 0, then adding 1 to the binary result.

+1 is

$\boxed{0}\boxed{00000000001}$

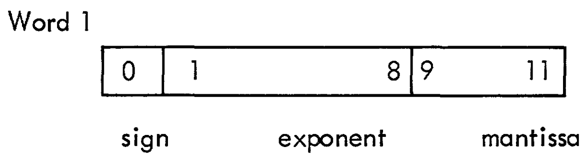
-1 is $\boxed{1}\boxed{11111111110}$ +1 = $\boxed{1}\boxed{11111111111}$ $= 7777_8$

The largest negative number is -2048 which is represented by 4000_8 or

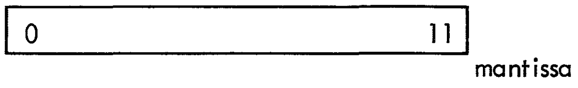
$\boxed{1}\boxed{00000000000}$

D.1.2 Real Numbers

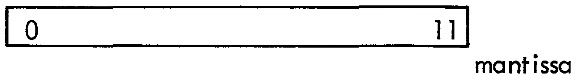
Real numbers are each allocated three machine words. They are represented as a binary mantissa multiplied by 2 raised to a binary exponent:



Word 2



Word 3



The sign of the number is bit 0 of word 1 (0=+, 1=-). The value and sign of the exponent are obtained by subtracting 10 000 000_2 (or 200_8) from bits 1 through 8 of word 1.

Example 1

110000001100
-0-
-0-

Sign:	1_2
Exponent:	10 000 0001_2
Mantissa:	$.100_2$

Exponent = $201_8 - 200_8 = 1_8$

Mantissa = $.4_8$

No. $= -.4_8 \times 2^{1_8}$

$= -1/2 \times 2 = -1$

Example 2

01000010110
-0-
-0-

Sign:	0_2
Exponent:	10 000 101_2
Mantissa:	$.1_2$

Mantissa = $.4_8$

Exponent = $205_8 - 200_8 = 5_8$

No. $= .4_8 \times 2^{5_8}$

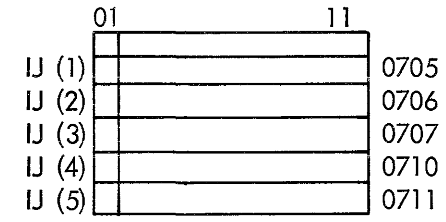
$= 1/2 \times 32 = 16$

D.2 STORAGE OF ARRAYS

Array variables are stored in core according to USA Standards, in columns and from top to bottom. For example, the array IJ

DIMENSION IJ (5)

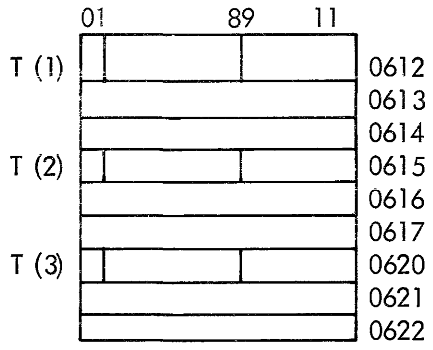
if started at location 0705 would be stored:



The real array, T

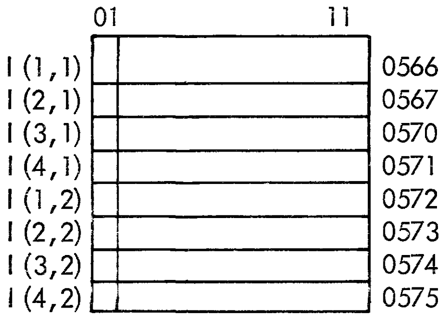
DIMENSION T (3)

starting in location 0612 would appear:



Two-dimensional arrays are stored as shown below.

DIMENSION I(4,2)



In the array

A(M(J,K))

M is a two-dimensional integer array stored as indicated above. No element of M may be less than 1.

If the element

M(3,4)

contains the integer 7, then A(M(3,4)) will be evaluated as A(7). The largest integer stored in M must not exceed the dimensions of A.

D.2.1 Representation of N-dimensional Arrays

Although arrays of more than two dimensions are illegal, the values of the subscripts of larger arrays may be calculated by using the following algorithm:

$$i_1+D_1*(i_2-1)+D_1*D_2(i_3-1)+\ldots D_1*D_2\ldots D_n(i_n-1)$$

where the subscript values are i_1, $i_2\ldots i_n$ in an array whose dimensions are D_1, $D_2\ldots D_n$.

Subprograms may be written to compute and insert subscript values in such illegal arrays. For example, in an array A(3,4,5), the following subprogram inserts the value of element A(N1,N2,N3):

```
      DIMENSION ARRAY (60)
      READ (1,5) N1,N2,N3, VALUE
      I=N1+3*(N2-1)+3*4*(N3-1)
      ARRAY(I)=VALUE
  5   FORMAT (3I1,F5.3)
      END
```

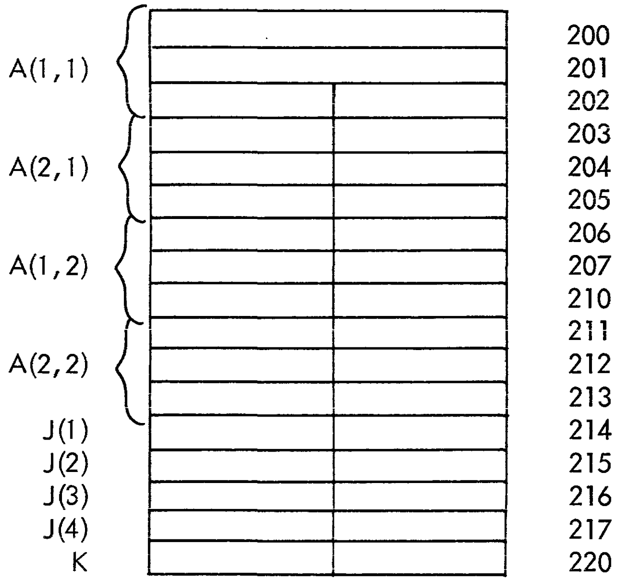
D.3 COMMON STORAGE ALLOCATION

Common storage begins in absolute location 200 in field 1. Variables are assigned locations in the common storage area in ascending order as they appear in COMMON statements.

For example:

```
COMMON A,J,K
DIMENSION A(2,2),J(4)
```

would be stored as follows.



NOTE

K does not appear in a DIMENSION statement.

If the COMMON statement of another subprogram defines

```
COMMON J
DIMENSION J(5)
```

J(1) through J(5) will be assigned to locations 1000 through 1004 respectively, thus overlapping the variables A(1,1) and A(2,1). The Loader is not aware of this, therefore it is advisable to make COMMON statements identical in all subprograms in which they appear.

However, the statements

```
COMMON DUMMY,J
DIMENSION DUMMY(2,2),J(4)
```

would not produce overlapping common and could be used in subprograms. In the example above, DUMMY is an arbitrary variable which need not be used in the subprogram.

APPENDIX E
ERROR MESSAGES

E.1 COMPILER

When an error is encountered during compilation of a statement, the incorrect statement and an error message is printed. Further compilation of that statement is terminated, and output is suppressed for the rest of the compilation. The compiler, however, will scan the remaining statements for errors, and will print an error message for any errors found.

An example of an error message follows:

```
/       A=B+M(6) + N(1)
                 ↑
MIXED MODE EXPRESSION
```

Note that an ↑ was printed directly below the incorrect statement. This indicates that the error occurred somewhere between that point and the beginning of the statement. In some cases the arrow may point directly at the illegal character or word, but this cannot always be assumed.

If an error occurs in the middle of a series of continuation lines, all remaining lines in that statement will be printed with the error message ILLEGAL CONTINUATION.

Compiler error messages are self-explanatory:

ILLEGAL CONTINUATION	ILLEGAL VARIABLE
ILLEGAL ARITHMETIC EXPRESSION	ILLEGAL OR EXCESSIVE DO NESTING
ILLEGAL STATEMENT	ARITHMETIC EXPRESSION TOO COMPLEX
ILLEGAL CONSTANT	MIXED MODE EXPRESSION
ILLEGAL STATEMENT NUMBER	EXCESSIVE SUBSCRIPTS
SYMBOL TABLE EXCEEDED	ILLEGAL EQUIVALENCING
SYNTAX ERROR (usually illegal punctuation)	

E.2 SABR

Because SABR is a 1-pass automatic paging assembler for binary relocatable programs, errors are somewhat difficult to handle. If there are errors in the source, the assembled binary code will be virtually useless. Both errors E and S are fatal. Assembly halts when they are encountered. The other types of errors are not fatal, but they cause the line in which they occur to be treated as a comment and thus essentially ignored. An address label on such a line will remain undefined and no space is reserved in the binary output for the erroneous data.

During the assembly pass error diagnostics are typed on the teletype as they occur.

Example:

```
C          AT          \10          +0004
```

This means that an error of type C has occurred at the 4th instruction after the location tag \10. This would correspond to statement 10 in the source program.

During the listing pass the error letter is typed in the address field of the instruction line.

The following error diagnostics may occur.

A	means that too many or too few ARG's follow a CALL statement.
C	means that an illegal character appears on the line. This could possibly be an "8" or "9" in an octal digit string or an alphabetic character in a digit string.
M	means that a symbol is multiply defined. It is impossible to resolve multiple definitions during Pass 2. Therefore, listings of programs which contain multiple definitions will necessarily have unmarked errors. The M flag occurs only during Pass 1.
I	means that an illegal syntax has been used. Below are listed the types of illegal syntax that may occur. (1) A pseudo-op with improper arguments. (2) A quote mark with no argument. (3) A non-terminated text string. (4) A memory reference instruction with improper address. (5) An illegal combination of microinstructions.
E	means there is no END statement.
S	means either one of two things: (1) The symbol table has overflowed. This can be corrected by using fewer symbols, using more shorter symbols, or by breaking the program into smaller parts. (2) Common storage has been exhausted.

One further type of error may occur. This is an undefined symbol. Because SABR is a one-pass assembler, undefined symbols cannot be determined until the end of the assembly pass, so the error diagnostic UNDF is given in the symbol table listing.

E.3 LINKING LOADER

If during the process of loading a program or subprogram the Linking Loader encounters an error, the user is notified by an error message; the partially loaded program or subprogram is ignored, removed from the field, and core is freed. The error messages are typed out in the form

ERROR 000n

where n is the error code number.

Error Code	Explanation
1	More than 64_{10} subprogram names have been seen by the Loader (64_{10} subprogram names is the capacity of the Loader's symbol table).
2	The current field is full.
3	The current subprogram has too large a COMMON storage assignment. (Subprogram with largest common storage declaration must be loaded first.) This is a semi-fatal error. Re-initialize the Linking Loader as explained below and reload the programs in the proper order.
4	Checksum error on input tape. If the error persists, re-assembly is necessary.
5	Illegal Relocation Code has been encountered. This can occur only if the relocatable binary tape is bad or if the user is using it improperly, e.g., not starting at the beginning of the tape, or reader error, or punch error. If the error persists, re-assembly is necessary.

Recovery from Errors 2, 4, 5 is accomplished by repositioning the tape in the reader to the leader code at the beginning of the subprogram and then pressing CONTinue. When attempting to recover from one of these errors, no other program should be loaded before reloading the program which caused the error. Obviously, on Error 2 a different field should be selected before pressing CONTinue.

The entire loading process may be restarted via the console switches, at any time by re-initializing the Linking Loader. To do this, set the console switches as follows: Data Field = h (the field where the Linking Loader resides), Instruction Field = h, Switch Register = 6200; then press START.

E.4 LIBRARY PROGRAM

During execution the Library programs check for certain errors and type out the appropriate error messages in the form

"XXXX" ERROR AT LOC NNNN

where XXXX specifies the type of error, and NNNN is location of the error. When an error is encountered, execution stops, and the error must be corrected.

When multiple error messages are typed, the location of the last error message is relevant to the user program. The other error messages are to subprograms called by the statement at the relevant location.

Error Code	Explanation
"ALOG"	Attempt to compute log of negative number
"ATAN"	Result exceeds capacity of computer
"DIVZ"	Attempt to divide by 0
"EXP"	Result exceeds capacity of computer
"FIPW"	Error in raising a number to a power
"FMT1"	Multiple decimal points
"FMT2"	E or . in integer
"FMT3"	Illegal character in I, E, or F field
"FMT4"	Multiple minus signs
"FMT5"	Invalid FORMAT statement
"FLPW"	Negative number raised to floating power
"FPNT"	Floating-point error may be caused by: Division by zero; floating-point overflow; attempting to fix too large a number
"SQRT"	Attempt to square root a negative number

To pinpoint the location of a Library execution error:

a. From the storage map, determine the next lowest numbered location (external symbol) which is the entry point of the program or subprogram containing the error.

b. Subtract in octal the entry point location of the program or subroutine containing the error from the LOC of the error in the error message.

c. From the assembly symbol table, determine the relative address of the external symbol found in step a and add that relative address to the result of step b.

d. The sum of step c is the relative address of the error, which can then be compared with the relative addresses of the numbered statements in the program.

APPENDIX F
OPERATING PROCEDURES

This appendix is a condensation of Chapter 8. The figures referenced (in parentheses) are found in the PDP-8/I System User's Guide, DEC-08-NGCB-D.

F.1 LOADING THE FORTRAN COMPILER

1. Load RIM and BIN Loaders into Field 1 (Figure RIM-1, 2, 3, and BIN-1).

2. Load the FORTRAN Compiler using BIN (Figure BIN-2); IF=1 SR=7777. When loaded, parts of the Compiler will be in Field 0 and Field 1.

To load the Compiler on the disk, proceed in step sequence, otherwise, proceed at step 5, below.

3. With the Disk Monitor in memory, call the Disk System Loader by typing:

```
.LOAD
```

and load the FORTRAN Compiler onto the disk (see PDP-8/I Disk/DECtape Monitor System, DEC-D8-SDAB-D.)

4. Save the compiler by typing:

```
.SAVE FTC0!0-7577: 5363
.SAVE FTC1!200, 1000-1577, 2600, 6000-16377;
```

F.2 COMPILING (Pass 1)

5. Set DF=0, IF=1, SR=1000

6. Place FORTRAN source program tape in reader, press punch ON, LOAD ADD, and then START; compilation commences.

7. Error message? Either proceed or correct program and recompile.

Compiler will punch compiled tape in two sections, separated by a noticeable length of blank tape.

8. More source program tapes to be compiled? Yes: insert source tape and press CONT. No: proceed to next step.

NOTE

The FORTRAN Compiler may be restarted at any time by pressing STOP and proceeding at step 5.

F.3 LOADING THE SABR ASSEMBLER

9. Load the SABR Assembler using BIN (Figure B-2); IF=1, SR=7777. When loaded, parts of the Assembler will be in Field 0 and Field 1.

To load the Assembler on the disk, proceed in step sequence, otherwise, proceed at step 12, below.

10. Same as step 3, above.

11. Save the Assembler by typing:

```
.SAVE SABR!0-7177; 200
.SAVE SAB1!12000 - 12427;
```

F.4 ASSEMBLING (Pass 2)

See Section 8.4 for alternate methods of assembling.

12. Insert Section 1 (the last section punched) of the compiled tape into the tape reader.

13. Set DF=0, IF=0, SR=0200, press LOAD ADD, START, and answer SABR's initial dialogue.

14. Turn the appropriate punch and reader ON; the tape reads in and the binary tape is punched.

15. Insert Section 2 (the first section punched) of the compiled tape into the tape reader and press CONT; assembly is completed when SABR halts after producing the relocatable binary tape.

SABR may be restarted to assemble another program by starting over at step 12 above.

SABR may be restarted at any time by pressing STOP, setting the SR=0200, and pressing LOAD ADD and then START.

To generate an assembly listing, proceed in step sequence, otherwise, proceed at step 18 below.

16. Insert Section 1 of the compiled tape into the reader and press CONT.

17. Insert Section 2 of the compiled tape into the reader and press CONT.

F.5 LOADING THE LINKING LOADER

18. Set DF=highest field in the configuration, IF=1, SR=7777, and press LOAD ADD.

19. Insert Linking Loader tape into the appropriate reader: if ASR reader, turn reader ON; if high-speed reader, set SR=3777.

20. Press START; the Linking Loader will be read into core memory.

F.6 LOADING PROGRAMS AND SUBPROGRAMS

21. Set DF and IF=to DF in step 18 above, SR=0200, and press LOAD ADD.

22. Insert relocatable binary tape (first, program or subprogram with largest amount of COMMON storage) into the reader with leader code over reader head.

23. Set SR as explained in Section 8.5.

24. Press START; the relocatable binary program will be loaded into core memory.

Repeat from step 22 for subsequent program or subprogram tapes or select an option (Core Availability or Storage Map) as explained in Section 8.5.

F.7 EXECUTING THE FORTRAN PROGRAM

25. Set DF and IF=to field of MAIN program, and SR=to starting address of MAIN program (determined from the Storage Map).

26. Turn punch ON and/or insert data tape in reader, as required.

27. Press LOAD ADD and START.

Program execution will begin.

APPENDIX G
IMPLEMENTATION NOTES

G.1 INPUT/OUTPUT

G.1.1 Implied DO Loops

Because of core memory restrictions, 8K FORTRAN does not have implied DO loops in READ and WRITE statements. However, a simple way to circumvent this restriction has been implemented. Normally a carriage return/line feed (CR/LF) is produced at the end of each WRITE statement. The CR/LF can easily be suppressed by terminating the WRITE statement with a comma. The CR/LF can be generated explicitly in one of two ways:

a. By using a WRITE (d,f) instruction.

b. By using a FINI pseudo instruction.

The second method is more efficient since it generates only 4 words of code, whereas the first method will generate somewhat more than that. For example, the following statements:

```
    DO 10  J=1,M
10  WRITE (1,20) (A(J,K), K=1,N)
20  FORMAT (10F7.3)
```

which is <u>not</u> legal in 8K FORTRAN, could be rewritten as follows:

```
    DO 15  J=1,M
    DO 10  K=1,N
10  WRITE  (1,20)  A(J,K),
15  WRITE  (1,20)
20  FORMAT  (F7.3)
```

or

```
    DO  15  J=1,M
    DO  10  K=1,N
10  WRITE  (1,20)  A(J,K),
15  FINI
20  FORMAT  (F7.3)
```

The second method is preferred for more efficient utilization of core memory. Note that it is not necessary to specify a repetition count in the FORMAT statement since the I/O handler initializes itself to the beginning of the FORMAT statement each time the WRITE statement is executed.

G.1.2 FORMAT Handling

For more complicated FORMAT handling a somewhat different technique can be used. For example,

```
      WRITE (1,20)  (A(K), K=1,N)
20    FORMAT (F7.2,2E15.6)
```

which again is not legal in 8K FORTRAN, could be written as follows

```
      WRITE  (1,20),          (comma suppresses CR/LF)
      DO 10  K=1,N
10    CALL IOH(A(K))
      FINI
20    FORMAT  (F7.2,2E15.6)
```

In the example above, the statement WRITE (1,20), generates the following assembly code

```
CALL 2, WRITE
ARG  (1
ARG  \20
```

The statement CALL IOH (A(K)) will generate code to call the subscripting routine SUBSC and will then generate the following code

```
CALL 1, IOH
ARG [0
```

where [0 is a temporary location generated by the compiler. Finally the FINI pseudo instruction will generate the following

```
CALL 1, IOH
ARG 0
```

which will cause execution of the WRITE statement to be completed.

Although only WRITE statements have been shown in the previous examples, the same techniques apply equally well to READ statements.

G.1.3 Numeric Input Conversion

In general, numeric input conversion is compatible with most other FORTRAN processors. A few exceptions are listed below:

a. Blanks are ignored except to determine what field digits fall in. Thus numbers are treated as if they were right justified within a field. In an F5.2 format, the following

```
bbb12
12bbb
.12bb
00012
```

would all be read as the number 0.12.

b. A null line delimited by two CR/LFs will be treated as a line of blanks, and blanks will be appended to the right of a line (if necessary) to fill out a FORMAT statement. Thus

```
12(CR/LF)
12bbb
bbb12
```

would all be identical under an F5.2 format. If an entire line is blank, numeric data from that line will be read as zeros.

c. No distinction is made between E and F format on input. Thus

```
100.
100E2
1.E2
10000
```

would all be read identically under either an F5.2 or E5.2 format.

G.1.4 Alphanumeric Data Within FORMAT Statements

Alphanumeric data may be transmitted directly from the FORMAT statement by two different methods: H-conversion or the use of single quotes.

Hollerith (H) format is used in WRITE statements only. An attempt to use H format specifications with a READ statement will cause characters from the format field to be either typed or punched. This may occasionally be a useful feature since it provides a simple way of identifying data that is to be read from the Teletype. For example, the following instructions

```
        READ (1,30) A,B
30      FORMAT (4HAb=b, F7.2/4HBb=b,F7.2)
```

would cause a =and B = to be typed out before the data was read.

The same effect is achieved by merely enclosing the alphanumeric data in single quotes. The result is the same as in H-conversion; on input, the characters between the single quotes are replaced by input characters, and, on output, the characters between the single quotes (including blanks) are written as part of the output data. For example, when referred to from a WRITE statement,

```
FORMAT ('PROGRAM COMPLETE')
```

would cause PROGRAM COMPLETE to be printed. This method eliminates the need to count characters.

G.1.5 E and F Format

When using the WRITE statement with either E format or F format with numbers less than 1.0 a zero will not be typed to the left of the decimal point.

G.2 ARITHMETIC OPERATIONS

G.2.1 Floating-Point

In general, floating-point arithmetic calculations are accurate to seven digits with the eighth digit being questionable. Subsequent digits are not significant even though several may be typed to satisfy a field width requirement.

No definitive information is currently available on the accuracy of the functions except that they are believed to be accurate to six decimal places for arguments which are neither extremely large nor extremely small.

The floating-point arithmetic routines check for both overflow and underflow. Overflow will cause the FPNT error message to be typed and program execution will be terminated. Underflow is detected but will not cause an error message. The arithmetic operation involved will yield a zero result. The arctangent function is accurate to six decimal places for arguments whose absolute value is greater than .01. This is a temporary restriction.

G.2.2 Integer

Integer arithmetic operations do not check for overflow. For example, the sum 2047+2047 will yield a result of -2. For more information refer to Chapter 1 of Introduction to Programming (Small Computer Handbook Series) or any text on binary arithmetic.

G.2.3 Exponentiation

Zero raised to a power of zero will yield a result of 1. Zero raised to any other power will yield a zero result. Numbers are raised to integer powers by repetitive multiplication. Numbers are raised to floating-point powers by calling the EXP and ALOG functions. A negative number raised to a floating-point power will not cause an error message but will use the absolute value. Thus, the expression (-3.0)**3.0 will yield a result of +27.

G.3 SUBSCRIPTING

Since excessive subscripting tends to use core memory inefficiently, it is suggested that subscripted variables be used judiciously. For example, the statement

```
A = ((B (I) + C2) * B(I) + C1) *B(I)
```

could be rewritten with a considerable saving of core memory as follows:

```
T = B(I)
A = ((T + C2) *T + C1)  *T
```

G.4 DO LOOPS

DO loops are treated slightly differently in 8K FORTRAN than in most compilers. The index is tested before the range of the DO is executed. Therefore, in the following example

```
       DO 20  N = 1, M
               .
               .
               .
    20  CONTINUE
```

the instruction between the DO statement and statement 20 will never be executed if M is less than one.

G.5 PAUSE STATEMENT

The PAUSE statement may be used for a variety of reasons to temporarily suspend program execution. In some cases the PAUSE statement may be used to give the operator a chance to change data tapes or to remove a tape from the punch. When this is done it is necessary to follow the PAUSE statement with a call to the OPEN subroutine. This subroutine initializes the I/O devices and sets hardware flags that may have been cleared by pressing the tape feed buttons. Example:

```
PAUSE
CALL OPEN
```

G.6 EQUIVALENCE STATEMENT

Because of core memory restrictions within the compiler, variables may not appear in EQUIVALENCE statements more than once. Thus,

```
EQUIVALENCE (A,B,C)
```

would be valid, but the statement

```
EQUIVALENCE (A,B), (B,C)
```

would not compile correctly.

G.7 SPECIAL I/O DEVICES

I/O can be performed on devices other than Teletype and high-speed paper tape reader and punch in several different ways:

1. If it is desired to use other devices in place of the high-speed paper tape reader and punch, rewrite the Utility library subroutine defining the entry points for the desired input and output devices as HSIN and HSOUT respectively. The source tape for the Utility subroutine is available from the program library and is very short. Refer to the SABR manual for more information.

2. If it is desired to input or output on a special device but not in ASCII format, write a subroutine to handle the particular device in the SABR assembly language. For more information refer to the SABR manual.

3. If it is desired to add additional devices which can be used with READ and WRITE statements, then edit part I of the Library Subroutines IOH. New entries must be made in the device transfer table at the beginning of IOH. Copies of this source tape and listings of the library subroutines are available from the program library. The service routines for the additional I/O devices must be written in SABR assembly language and can then be assembled along with the revised version of IOH.

4. Program written in SABR language can call PAL subroutines in various ways:

   a) A JMS 7000 instruction will call a PAL program which starts at location 7000 in the same memory field.

   b) A CONTINUE (or PAUSE) statement might be inserted in the user's FORTRAN program. Then a JMS to the PAL subroutine may be inserted using the Switch Register.

It is possible to load any size PAL III program for linkage with an 8K FORTRAN program by merely dimensioning an integer variable to the proper size for the PAL III program. This offers two advantages, virtually unlimited size programs in PAL III can be linked to 8K FORTRAN main programs, and none of the library routines are disturbed by this linkage.

G.8 ERRORS

All compile time, assembly time, and execution time errors are fatal. For this reason it is desirable to suppress punched output of the compiler and assembler until the source program is believed to be correct. For specific instructions refer to Chapter 8.

Note especially that the compiler will not detect undefined statement numbers. Therefore it is important to examine the assembly symbol table for undefined symbols before loading and executing the program.

Do not attempt to load or run a program which has assembly errors. Do not attempt to proceed after an execution time error by pressing CONTinue. Unpredictable results will be obtained in either case.

HOW TO OBTAIN REVISIONS AND CORRECTIONS

Notification of new or revised DEC software and manuals available from the Program Library is published in:

Digital Software News for the PDP-8 Family
Digital Software News for the PDP-9 Family

If you are not receiving the publication appropriate to your computer, please notify Software Information Service (see Reader's Comments card).

Revised software products and documents are shipped only after the Program Library receives a specific request from a user (see title page for address).

Digital Equipment Computer User's Society (DECUS) maintains a library of user software and publishes them in DECUSCOPE, a magazine available to both DECUS members and to non-members who request it. Return the request card below to receive further information or to place your name on the mailing list.

To: Decus Office,
Digital Equipment Corporation,
Maynard, Massachusetts 01754

☐ Please send DECUS installation membership information.

☐ Please send DECUS individual membership information.

☐ Please add my name to the DECUSCOPE non-member mailing list.

Name ______________________

Company ______________________

Address ______________________

____
(Zip Code)

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback -- your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability.

Did you find errors in this manual? Please explain, giving page numbers.

How can this manual be improved?

DEC also strives to keep its customers informed on current DEC software and publications. Thus, the following periodically distributed publications are available upon request. Please check the publication(s) desired.

- ☐ Digital Software News for the PDP-8 Family, contains current information on software problems, programming notes, new and revised software and manuals.
- ☐ PDP-8/I Software Manual Update, contains addenda/errata sheets for updating software manuals.
- ☐ PDP-8/I User's Bookshelf, contains a bibliography of current and forthcoming software manuals.

Please describe your position.

Name		Organization	
Street		Department	
City	State	Zip or Country	

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

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

Postage will be paid by:

digital

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