digital
software

OS/8

Language Reference Manual

Order No. AA-H609A-TA

OS/8

Language Reference Manual

Order No. AA-H609A-TA

ABSTRACT

This document describes the following languages supported by OS/8: BASIC, FORTRAN IV, PAL8, FORTRAN II, FLAP/RALF, SABR.

SUPERSESSION/UPDATE INFORMATION: This manual supersedes sections of the OS/8 Handbook (DEC-S8-OSHBA-A-D).

OPERATING SYSTEM AND VERSION: OS/8 V3D

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, March 1979

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

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	

CONTENTS

BASIC

FORTRAN IV

PAL8

FORTRAN II

FLAP/RALF

SABR

DOCUMENTATION SET FOR OS/8

OS/8 SYSTEM GENERATION NOTES (AA-H606A-TA)

The System Generation Notes provide the information you need to get a new OS/8 system running.

OS/8 SYSTEM REFERENCE MANUAL (AA-H607A-TA)

The System Reference Manual describes OS/8 system conventions, keyboard commands, and utility programs.

OS/8 TECO REFERENCE MANUAL (AA-H608A-TA)

The TECO Reference Manual describes the OS/8 version of this character-oriented text editing and correcting program.

OS/8 LANGUAGE REFERENCE MANUAL (AA-H609A-TA)

The Language Reference Manual describes all languages supported by OS/8, including BASIC, FORTRAN IV, and the PAL8 assembly language.

OS/8 ERROR MESSAGES (AA-H610A-TA)

This manual lists in alphabetical order all error messages generated by OS/8 system programs and languages.

BASIC

CONTENTS

CONTENTS (Cont.)

FIGURES

TABLES

CHAPTER 1

OS/8 BASIC

1.1 OVERVIEW

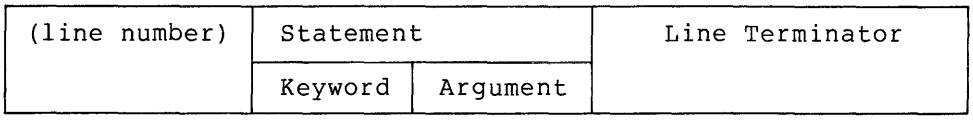
BASIC (Beginner's All-Purpose Symbolic Instruction Code) is a high-level computer language for scientific, commercial, and educational applications.

- BASIC is all-purpose. You can use it to process large amounts of data as well as to solve complex mathematical problems.
- BASIC is conversational. You write programs with simple English keywords and common mathematical expressions. You run, store, and retrieve programs with a set of simple commands resembling English verbs.
- BASIC is interactive. You can input data while a program is running and make changes and corrections in statements under the direction of the BASIC editor. BASIC locates any formatting errors you make in entering your program and prints appropriate messages to help you correct them.

1.1.1 Writing a BASIC Program

You write a BASIC program as a series of numbered lines, each containing one or more instructions called statements.

The format of a typical one-line statement line is



For example:

```
10              PRINT     "HOORAY!"     RETURN
```

The line number -- which may range from 1 to 99999 -- identifies the line and indicates its position in the sequence of operations set out in the program. You do not have to enter the lines in numerical order. BASIC automatically sorts them before it executes the program. You may remove or insert lines at any time to modify a program. For this reason, it is good programming practice to leave room for later additions by numbering lines in increments of five or ten.

The first element in the statement -- the keyword -- tells BASIC what to do. For example, the keyword in the example -- PRINT -- instructs BASIC to display output, such as a message or the result of a computation, on the terminal.

The second element in the statement -- the argument -- may be a formula, a word or phrase, a variable, a line number -- anything BASIC can take action upon. In the example above, the argument is the message "HOORAY!", which BASIC will display on the terminal.

The line terminator -- RETURN -- enters the program line into the system. Even though you type the line on the keyboard and see it echoed on your terminal screen, the BASIC editor does not receive it until you strike the RETURN key.

The last line in every BASIC program must be an END statement. The format is

(line number) END

BASIC statements are described in Sections 1.3 through 1.7.5.

1.1.2 The BASIC Character Set

The alphabet of the BASIC language is the full set of ASCII (American Standard Code for Information Interchange) characters. This set includes

- Upper-case letters A through Z
- Numbers 0 through 9
- Special characters (* and $, for example)
- Nonprinting characters (space and tab, for example)

You may include all ASCII characters in a program. BASIC converts lower-case letters to upper case, ignores nonprinting characters, and leaves all other characters unchanged.

1.1.3 Entering and Running a BASIC Program

To run a BASIC program you must first enter it into a special area in the memory of your computer -- called the workspace -- that BASIC reserves for user-written programs. To do this, summon the BASIC editor by typing

```
.BASIC
```

in response to the Monitor dot. BASIC will then display the message

```
NEW OR OLD --
```

to determine if you want to enter a program from the terminal or run one that you have previously stored as a file.

Assume, for example, that you have a new program called CHEERS that you want to enter and run. Type

```
NEW OR OLD -- NEW CHEERS
```

As soon as BASIC displays a message to indicate that it is READY, begin typing your program line by line.

```
10 FOR K=1 TO 3
20 PRINT "HOORAY!"
30 NEXT K
99 END
```

This complete four-line program now resides in the workspace. To run it, type the command

```
RUN
```

BASIC displays a header line, followed by the program output.

```
HOORAY!
HOORAY!
HOORAY!
```

In addition to the RUN command, BASIC provides commands that let you display on the terminal the program that is in the workspace, store it as a file on a peripheral device and retrieve it later for re-use, change its name, renumber it, or erase it from the workspace. For a complete description of BASIC commands, see Section 1.11.

1.2 ELEMENTS OF BASIC

The following sections define the elements of BASIC programming.

1.2.1 Constants

A constant is a quantity with a fixed value. In BASIC, you may enter constants from the terminal or instruct BASIC to read them from a data list or from a file during program execution.

1.2.1.1 **Numeric Constants** - BASIC accepts numbers within the range

```
10^-616<N<10^+616
```

and treats all numbers as decimal numbers. That is, it accepts any number containing a decimal and assumes a decimal point after any integer.

BASIC uses a second format -- called exponential or E-type notation -- to express numbers outside the range +.00001<N<999999. The format for an E-type number is

```
xxxx.xxxx E(+ or -)nnn
```

where E represents "times 10 to the power of." Thus, for the number 23.4E2, read "23.4 times 10 to the power of 2." Expressed another way,

```
23.4E2=23.4*10**2=2340
```

You may input data in either format. Results of computations with an absolute value outside the range +.00001<N<999999 are always output in E-type format.

BASIC prints six significant digits in normal operation as shown in the following examples.

```
You enter:      BASIC outputs:
.01             .01
.0099           .0099
999999          999999
1000000         1.00000E+006
.0000009        9.00000-007
```

BASIC automatically suppresses leading zeros in integer numbers and trailing zeros in decimal fractions. BASIC outputs exponential numbers in the form

```
(blank or -) x.xxxxxE(+ or -)nnn
```

For example:

```
-3.37021E+008 equals -337,021,000
 7.26000E-004 equals 0.000726
```

BASIC stores numbers internally with a precision of 23 bits. Arithmetic operations are accurate to 22 bits. No rounding is done.

BASIC does conversions from ASCII to internal format and vice-versa in extended precision. Conversion to internal format is rounded to 23 bits. On output, BASIC rounds the result to 6 decimal digits.

1.2.1.2 **String Constants** - A string constant is any keyboard character or group of characters -- letters, numbers, spaces, symbols -- that you want to use as data. In BASIC programs, string constants must be enclosed by quotation marks. The quotation marks instruct BASIC to treat characters within them exactly as you type them in at the terminal.

For example, this program of PRINT statements

```
10 PRINT "I AM A STRING"
20 PRINT "@%$^*& $%&"
30 PRINT "$346.98"
40 PRINT """HI THERE!"""
50 PRINT "30 + 20"
60 PRINT 30 + 20
99 END
```

will cause BASIC to display

```
I AM A STRING
@%$^*& $%&
$346.98
"HI THERE!"
30 + 20
 50
```

Note that BASIC does not consider the enclosing quotation marks to be part of the string. As line 40 demonstrates, to display quotation marks, you must place them within a double pair.

Lines 50 and 60 show the difference between string and numeric data. The quotation marks cause BASIC to display the string "30 + 20" exactly as you enter it. In line 60 BASIC performs a computation on the expression 30 + 20 and prints the sum of 50.

1.2.2 Variables

In BASIC programming, a variable is a symbolic name representing a number or a character string. When you assign a numeric or string value to a variable (with a LET statement, for example), the value is said to be "stored" in the variable. This means that BASIC has placed the value in a memory location -- or locations -- associated with the variable name.

For example, the following statement stores the value 37 in the variable N.

```
LET N=37
```

If N already contains a value, the new value replaces it.

Once you have assigned a value to a variable, BASIC will use the value in any expression in which the variable appears. For example:

```
LET B=N*2
```

This statement evaluates the expression N*2 and stores the result in the variable B.

You may instruct BASIC to change the value of a variable any number of times during one execution of a program. BASIC always uses the most recently assigned value when performing calculations.

The following sections describe numeric variables, string variables, and subscripted variables.

1.2.2.1 **Numeric Variables** - A numeric variable name consists of a letter or a letter followed by a digit. For example,

Acceptable Variables	Unacceptable Variables
M	2C (A variable cannot begin with a digit.)
R2	AB (A variable may contain only one letter.)

Unless you specify otherwise, BASIC automatically sets all variables to zero before executing a program. However, if you wish to assign zero, it is good programming practice to do the initializing yourself at the beginning of the program. You can do this with a series of LET statements or by using READ and DATA statements. For example, this statement

```
10 LET A=0\B=0\C=5
```

tells BASIC to assign 0 to A and B and 5 to C. (See Section 1.6.2 for READ and DATA.)

1.2.2.2 **String Variables** - A string variable name consists of a letter -- or a letter and a digit -- followed by a dollar sign. A$ and A2$ are both legitimate string variable names; 2A$ and AA$ are not.

You may assign no more than eight characters to a string variable unless you have first specified the dimensions with a DIM statement. (See Section 1.6.3.)

1.2.2.3 **Subscripted Variables** - A subscripted variable consists of a string or numeric variable name followed by a subscript in parentheses. A subscript may be a number, a numeric variable, an expression, or any two such elements separated by a comma. The following are all legal subscripted variables:

```
A(2)

A(K)

M(3,4)

M(I,J)

G(K-1)

F$(3.4)
```

If the subscript is not a whole number, BASIC uses only the whole number part. Thus, in the example above F$(3.4) is the same as F$(3). BASIC permits a subscript value of zero.

The subscript in a numeric variable serves as a pointer to a location in a list or table. For example, this subscripted numeric variable

```
X(3)
```

indicates the fourth position in the list

```
X(0), X(1), X(2), X(3), X(4), X(5)
```

Note that all subscripted variables in a list or table share the same variable name.

Two subscripts appended to a numeric variable (and separated by commas) indicate a row and column number in a table. This variable

```
A(3,5)
```

points to row three column five.

One subscript appended to a string variable name indicates the length of the string. Two subscripts indicate its position in a list and its length. For example, this variable

```
R$(35)
```

will accept a string constant 35 characters long. This variable

```
R$(5,35)
```

indicates that the sixth position in a list (beginning with 0) holds a string 35 characters long.

You cannot create a table of string variables in a BASIC program.

A program may contain the same variable in both a subscripted and an unsubscripted form. For example, BASIC will recognize A(1) and A in the same program. However, once subscripted, a variable must contain the same number of subscripts throughout the program. If A(1) occurs, for example, BASIC will not accept A(3,4).

For further discussion of lists and tables, see the DIM statement in Section 1.6.3.

1.2.3 Expressions

An expression is a group of numerics or alphanumerics which, when evaluated, equals a number or a string. Expressions contain special symbols -- called operators -- which direct BASIC in its evaluation. BASIC recognizes three types of operators:

- Arithmetic operators
- Relational operators
- String operators

1.2.3.1 **Arithmetic Expressions** - BASIC uses the following operators to perform addition, subtraction, multiplication, division, and exponentiation.

Symbol	Meaning	Example
+	Addition	A+B
-	Subtraction	A-B
/	Division	A/B
^ or **	Exponentiation	A/B or A**B

In any mathematical formula, BASIC first treats expressions enclosed by parentheses. After parentheses, BASIC maintains the following order of priority.

1. Exponentiation
2. Multiplication and Division (equal priority)
3. Addition and Subtraction (equal priority)

When all of the operators in an expression have equal claim to priority, BASIC simply evaluates the expression from left to right. For example, in this expression,

```
A+B-C
```

BASIC adds A to B and then subtracts C from the sum.

Parentheses let you control the order in which BASIC performs the operations called for in an expression. You may nest parentheses within parentheses. Where nesting occurs, BASIC will give first attention to the elements contained in the innermost "nest."

In this example,

```
A=7*((B**2+4)/X)
```

the order of priority is

1.	B**2	BASIC raises B to the power of 2.
2.	B**2+4	BASIC adds 4 to B**2.
3.	(B**2+4)/X	BASIC divides the result so far by X.
4.	7((B**2+4)/X)	BASIC multiplies by 7 and then assigns the result to A.

Since BASIC ignores spaces, you may use them to make complex expressions easier to read. Spacing will considerably improve the appearance of the example above.

```
A = 7*( (B**2 + 4)/X )
```

1.2.3.2 **Relational Expressions** - Relational operators instruct BASIC to determine the relationship between two values in an expression. BASIC recognizes six relational operators.

=	equal
<	less than
=< or <=	less than or equal to
>	greater than
=> or >=	greater than or equal to
<> or ><	not equal

Relational operators set the conditions in IF-THEN statements. This statement

```
10 IF A>B THEN 50
```

directs BASIC to determine the relationship between A and B and jump to line 50 if A is greater.

You can use strings and string variables in relational expressions. BASIC compares strings one alphanumeric character at a time, using ASCII code numbers to determine if one character is "greater" or "less" than another. BASIC proceeds from left to right until it reaches the end of the strings or until it discovers an inequality. If one string is shorter, BASIC adds spaces to it until both are the same length. For example, in comparing AB to ABCD, BASIC will treat AB as AB (space) (space).

1.2.3.3 **String Concatenation** - BASIC recognizes the ampersand (&) as an operator in string expressions. The ampersand allows you to concatenate strings -- that is, to join them together. For example:

```
10 LET A$="BEAN"
20 LET B$= "TOWN"
30 PRINT A$ & B$
99 END
```

This program will cause BASIC to display:

```
BEANTOWN
```

You may use the ampersand to concatenate strings wherever a string is legal -- with one exception. A concatenated string variable may not appear to the left of the equal sign in a LET statement. Thus, this statement is legal:

```
10 LET A$=B$ & C$
```

this statement is not:

```
10 LET A$ & B$ = C$
```

1.3 FORMATTING BASIC STATEMENTS

Every BASIC program consists of a sequence of numbered lines, each containing one or more instructions called statements.

The format of a typical single-statement line is

(line number) keyword argument

where the keyword is an instruction to BASIC and the argument is some element that BASIC can act upon.

Here are some examples of single-statement lines.

```
10 PRINT "HOORAY"

20 LET A = 8

45 GOTO 90

80 INPUT R$
```

A multistatement line is one that contains more than one keyword/argument combination. The format is

(line number) STATEMENT1\STATEMENT2\STATEMENT3

For example:

```
30  LET X=X + 1 \ PRINT X \ IF X=25 GOTO 80
```

BASIC executes the statements in a multistatement line from left to right. The backslash -- like RETURN -- terminates a statement.

The line number -- which may range from 1 to 99999 -- identifies the line and any statement or statements it contains; it also indicates a line's position in the sequence of operations set out in the program. Keep in mind the following features and rules when entering and numbering BASIC lines.

- You may enter lines in any order. The RUN command causes BASIC to sort all lines into numerical order before executing the program.
- You may add, delete, or shift lines at any time to modify your program.
- You should number lines in increments of five or ten, in order to leave room for additional statements you may want to insert later.
- If your modified program contains consecutively numbered lines, making it difficult to insert further statements, you may renumber your program with the BASIC RESEQ program. The RESEQ program lets you specify a suitable increment between lines.

The keyword -- the first element in the statement -- tells BASIC what it must do in order to successfully execute the instruction. The argument of the statement is the entity that BASIC acts upon. It may be a number, a string, an expression, a variable, or a line number. For example, in the single-statement lines above, the keyword PRINT tells BASIC to display the string HOORAY! on the screen, the keyword LET to assign the value 8 to the variable A, the keyword GOTO to jump to line 90, and the keyword INPUT to receive a value from the terminal and assign it to the variable R$.

1.4 THE ASSIGNMENT STATEMENT -- LET

The LET statement uses the equal sign (=) to assign a value to a variable.

The format is

(line number) [LET] v = expression

where

v	is a variable
expression	is a number, a string, a variable, or an arithmetic expression

The LET statement is the only BASIC statement in which the keyword is optional. For example, these two lines

```
10 LET A = 5
```

```
10 A = 5
```

will both cause BASIC to assign the value 5 to the variable A.

The equal sign in a LET statement indicates replacement rather than equality. That is, the LET statement causes BASIC to evaluate the expression on the right of the equal sign and assign the value to the variable on the left, replacing its previous value. For example, the statement

```
15 K = K + 1
```

causes BASIC to add one to the value of K and store the result in the variable K.

BASIC performs any mathematical operations and functions that you call for in a LET statement. In this statement

```
20 LET A = C + SQR(B)
```

BASIC sets the variable A equal to the value of C plus the square root of the variable B.

This statement

```
5 A$ = "YAZ"
```

assigns a string to a string variable.

The following statement causes BASIC to set element 3,2 in array A equal to element 1,4 in array B.

```
20 LET A(3,2) = B(1,4)
```

1.5 THE COMMENT STATEMENT -- REMARK

The REM statement lets you document your source program with notes and comments, for example:

```
5 REM SUBROUTINE SWAPS VALUES A AND B
```

1.6 INPUT AND OUTPUT STATEMENTS

BASIC provides you with three ways to supply a program with data:

- The INPUT statement lets you type in data while the program is running.
- The READ, DATA, and RESTORE statements let you insert data into the program before you run it.
- BASIC file statements make it possible for you to store data outside the main program and retrieve it under program control.

This section describes only the first two methods. See Section 1.9 for information on file input and output.

The BASIC PRINT statement causes BASIC to display strings and the results of computations on the terminal.

1.6.1 INPUT

The INPUT statement allows you to enter data while the program is running.

The format of the INPUT statement is

```
(line number) INPUT x1, x2,...,xn
```

where x1 through xn represent numeric variables or string variables. If the INPUT statement contains both numeric and string variables, you must enter the appropriate type of data in the proper sequence, assigning numbers to numeric variables and data strings to string variables.

For example, the following line

```
10 INPUT A, B$, C
```

requires a number, a string, and another number entered in that order.

The INPUT statement causes BASIC to pause during the execution of the program, print a question mark (?), and wait for you to type in one value for each variable in the statement. Enter the values, separating them with commas, and press the RETURN key. If you press RETURN without typing in all the data requested, BASIC will display another question mark and await the rest of the data. If you provide more data than the statement requests, BASIC saves the remaining or unused data for use by the next INPUT statement.

BASIC recognizes only the following characters as numeric data.

digits 0 through 9

+ or -

the letter E (for use in floating-point numbers)

. (first decimal point)

BASIC ignores leading spaces and treats all other characters as delimiters for separating numeric data. When BASIC encounters a character other than those specified above, it will assume that it has come to the end of the entry relating to the variable it is currently reading and will apply any character typed in after that to the next variable. Two delimiters in succession signify that the data between delimiters is 0.

For example, the following program requires five numbers:

```
10 INPUT A,B,C,D,E
.
.
.
99 END
```

BASIC prints a question mark to request data.

```
?
```

If, in response to the INPUT prompt, you type

```
-2, 3.7A4E3 9>+1
```

BASIC will assign values to variables in the following manner:

A:-2, B:3.7, C:4000 (4E3=4x10^3=4000), D:9, E:1

BASIC recognizes all characters -- including quotation marks -- as string data and assumes a string length of 8 characters unless you have defined the string variable with a DIM statement. (See Section 1.6.3.) Since it accepts all characters as string data, BASIC treats only the carriage return as the delimiter of a string. To terminate a data string, type the RETURN key.

1.6.2 READ, DATA, and RESTORE

The READ and DATA statements make it possible for you to include data to a program before you run it. During execution, BASIC assigns values listed in the DATA statement to the variables in the READ statement. READ and DATA statements occur only in combination with each other. RESTORE causes BASIC to reuse the values in a DATA statement.

The format of the READ statement is

(line number) READ x1, x2,..., xn

where x1 through xn represent variable names separated by commas.

The format of the DATA statement is

(line number) DATA x1, x2,...,xn

where x1 through xn represent values separated by commas.

Like the INPUT statement, the READ statement must occur in the program before the point where the data is required. DATA statements normally appear at the bottom of the program before the END statement, where you can find them easily when you wish to change input data.

BASIC handles the items in READ and DATA statements sequentially. That is, it assigns the first value in the DATA statement to the first variable in the READ statement, the second variable to the second value, and so on.

A READ statement may contain more or fewer variables than there are values in one DATA statement. READ causes BASIC to search all available DATA statements in the order of their line numbers until it has found values for all variables. When it has assigned values to all of the variables in one READ statement, BASIC will hold the remaining values in the DATA statement until it comes to the next READ statement.

All three of these routines will instruct BASIC to set variable A equal to 1, variable B equal to 2, and variable C equal to 3.

```
10 READ A,B,C
.
.
.
75 DATA 1,2,3
99 END
```

```
10 READ A,B,C
.
.
.
75 DATA 1,2
80 DATA 3
99 END
```

```
10 READ A
.
30 READ B,C
.
.
75 DATA 1,2,3
99 END
```

A DATA statement may contain both string and numeric data. String data in a DATA list must always be enclosed by quotation marks.

This program will cause BASIC to assign 5 to variable C, "AAA" to variable D$, 12 to variable E, and "BEER" to variable F$.

```
10 READ C, D$, E, F$
.
.
.
75 DATA 5, "AAA", 12, "BEER"
```

The RESTORE statement makes it possible for you to use the same data more than once in a program. RESTORE instructs BASIC to reset the data pointer to the first value in the first DATA statement in the program. Since BASIC then proceeds to read through the values as though for the first time, you may use the same variable names on the second pass through the data.

The following program reads a DATA list twice.

```
10 READ A,B,C,D
20 PRINT A;B;C;D
30 RESTORE
40 READ E;F;G;H
50 PRINT E,F,G,H
75 DATA 1,2,3,4
99 END
```

BASIC displays

```
 1  2  3  4
 1  2  3  4
```

1.6.3 DIMENSION

The DIM statement lets you create a list or table of subscripted variables for storing data. (You can organize numeric data in both lists and tables, but BASIC stores strings in lists only.) DIM also defines the length of a string assigned to a string variable.

To create a list -- a one-dimensional array -- of subscripted numeric variables, use the following format:

(line number) DIM x(n)

where

- x — is a numeric variable name. All subscripted variables in the list share the same variable name.
- n — specifies the number of numeric elements in the list. (Since BASIC assigns 0 as the subscript of the first variable, the number of elements in the list is n + 1.)

For a table -- a two-dimensional array -- of subscripted numeric variables, use the form

(line number) DIM x(n,m)

where

- x — is a numeric variable name
- n — specifies the number of rows in the table. (The actual number of rows in the table is n + 1.)
- m — specifies the number of columns. (m + 1 equals the number of columns.)

For example, this DIM statement introduces a list of six subscripted numeric variables:

```
10 DIM A(5)
```

10 DIM A(5)

A(0)	A(1)	A(2)	A(3)	A(4)	A(5)

The following statement describes a table of 24 numeric elements.

```
10 DIM A(3,5)
```

SIX COLUMNS

FOUR ROWS						
	A(0,0)	A(0,1)	A(0,2)	A(0,3)	A(0,4)	A(0,5)
	A(1,0)	A(1,1)	A(1,2)	A(1,3)	A(1,4)	A(1,5)
	A(2,0)	A(2,1)	A(2,2)	A(2,3)	A(2,4)	A(2,5)
	A(3,0)	A(3,1)	A(3,2)	A(3,3)	A(3,4)	A(3,5)

The number of elements in a table is (n + 1)*(m + 1).

To specify the length of a string, use the DIM statement in the following manner.

```
(line number) DIM X$(n)
```

where

X$ is a string variable

n is the length of the string. A string may contain no more than 72 characters. All strings that exceed 8 characters in length must be dimensioned with a DIM statement.

To introduce a list of subscripted string variables, use the format

```
(line number) DIM X$(n,m)
```

where

X$ is a string variable name

n specifies the number of strings in the list. (The number of strings is n + 1.)

m is the length of each string -- up to 72 characters

For example, this DIM statement describes one string 12 characters long:

```
10 DIM C$(12)
```

This statement describes 4 strings, each 20 characters long:

```
10 DIM D$(3,20)
```

This program will fill variables from a DATA list:

```
10 DIM D$(3,20)
20 FOR Y=0 TO 3
30 READ D$(Y)
40 NEXT Y
50 FOR Z=0 TO 3
60 PRINT D$(Z)
70 NEXT Z
80 DATA "ZERO","ONE","TWO","THREE"
```

Keep in mind the following features and rules concerning the DIM statement:

- Arrays are limited in size only by the amount of available memory -- that is, space not used by the monitor or the program statements.
- Subscripts n and m must be integer numbers. They may not be variables.
- A variable may not appear in a program with subscripts higher than the ones you have described in the DIM statement.
- BASIC assumes a string length of 8 characters or less unless you define the string variable with a DIM statement. If you wish to assign a string that is more than 8 characters long, you must DIMension the string variable.
- BASIC will not accept two-dimensional string variables.
- BASIC assigns a subscript of 0 to the first element in every array. Therefore, the number of elements in a one-dimensional array is n + 1, and the number of elements in a two-dimensional array is (n + 1)*(m + 1).
- You may define more than one array with a single DIM statement. For example, this statement dimensions both the one-dimensional array A and the two-dimensional array B.

```
10 DIM A(20), B(4,7)
```

1.6.4 PRINT

The PRINT statement lets you instruct BASIC to display the results of computations, comments, and the values of variables, or to plot the points of a graph on a terminal.

The format of the PRINT statement is

```
(line number) PRINT expression(s)
```

where expressions are numbers, variables, strings, or arithmetic expressions separated by format control characters. Using the PRINT statement without expressions will output a blank line on the terminal.

To output the result of a computation or the value of a variable at any point in the program, type the line number, PRINT, and the variable name or names separated by a format control character. BASIC will use the current value of the variables to evaluate any algebraic expression in a PRINT statement. Thus, the program

```
10 A=16 \B=5 \C=4
20 PRINT A; (C+B)/3; SQR (A)
99 END
```

will output the following values on the terminal:

```
16  3  4
```

To print a message or comment on the screen, type the text, enclosed by quotation marks, as the expression of a PRINT statement. Use PRINT message statements in combination with INPUT statements to specify the data to be entered.

```
10 PRINT "NUMBER OF SHEEP"
20 INPUT S
```

These lines in a program will produce the following output on the screen.

```
NUMBER OF SHEEP
?
```

PRINT statements may contain a combination of messages and numeric variables. This line

```
50 PRINT "TOTAL NUMBER OF SHEEP ="; T
```

will (assuming that T=354) cause the following to be output during execution of the program:

```
TOTAL NUMBER OF SHEEP = 354
```

1.6.4.1 **Printing Zones -- Format Control Characters** - OS/8 BASIC divides a terminal line into five fixed zones (called print zones) of fourteen columns each. To output data in a five-zone format, separate the variables in the PRINT statement with commas. To output data in a single-space row, separate the variables with semicolons.

The following program illustrates the use of control characters in PRINT statements:

```
10 READ A,B,C
15 PRINT A,B,C,A**2,B**2,C**2
20 PRINT
30 PRINT A;B;C;A**2;B**2;C**2
75 DATA 4;5;6
99 END
RUNNH
 4             5             6             16            25
 36

 4  5  6  16  25  36

READY
```

As this example illustrates, when you list more than five variables in a PRINT statement, BASIC automatically moves the sixth number to the beginning of the next line.

1.6.4.2 **Printing Numbers and Strings** - BASIC prints all numbers (integer, decimal, and E-type) in the following format:

```
sign number space
```

where the sign is either minus (-) or blank and the number is always followed by a blank space.

BASICBASIC prints strings exactly as you type them with no leading or trailing spaces. (To print quotation marks, you must delimit them with a double pair.)

For example:

```
10 PRINT """PRINTING QUOTATION MARKS"""
20 END
RUNNH

"PRINTING QUOTATION MARKS"
```

1.6.4.3 **Printing with the TAB and PNT Functions** - The TAB function allows you to position characters anywhere on the terminal line. You may use the TAB function only in combination with a PRINT statement.

The format of the TAB function is

```
TAB(X)
```

where X is the position (from 1 to 72 columns available on the terminal) in which the next character will be displayed.

Each time the TAB function appears in a PRINT statement, BASIC counts the positions from the beginning of the line, not from the current position of the printing head. For example, the TAB function in the following program causes BASIC to print the character "/" at 24 equally spaced positions across the line.

```
10 FOR K=3 TO 72 STEP 3
20 PRINT TAB(K);"/";
30 NEXT K
99 END
```

If the argument X in the TAB function is less than the current position of the printing head, BASIC starts printing at the current position. If the argument is greater than 72 (the number of columns available in an output line), BASIC executes a carriage return and a line feed and then resumes printing at position 1.

The PNT function allows you to perform special nonprinting actions on the terminal, such as ringing the buzzer, erasing the screen, moving the cursor, etc.

The format of the PNT function is

```
PNT(X)
```

where the argument X represents the decimal value of the 7-bit ASCII character to be output.

For example, to ring the buzzer on the terminal, type

```
10 PRINT PNT(07)
```

1.7 CONTROL STATEMENTS

During the execution of a program, BASIC ordinarily passes from one line to the next in ascending numerical order. BASIC control statements make it possible for you to alter the normal sequence -- either unconditionally or only when certain conditions are met. Thus, you can:

- repeat a set of statements
- skip statements
- stop and check values
- terminate a program

This section describes the statements that allow you to change the normal sequence of statement execution.

1.7.1 Unconditional Transfer -- GOTO

The GOTO (or GO TO) statement causes BASIC to jump to any line in the program that you specify. The GOTO statement sets no conditions.

The format of the GOTO statement is

```
(line number) GOTO n
```

where n is the number of the line to which BASIC will jump.

When BASIC encounters a GOTO statement, it jumps immediately to the line beginning with the number indicated. For example, this program

```
10 GOTO 40
20 PRINT "SECOND"
30 STOP
40 PRINT "FIRST"
50 GOTO 20
99 END
```

will display

```
FIRST
SECOND
```

If you specify a nonexecutable statement (such as REM) in a GOTO line, BASIC will proceed to the next executable statement.

NOTE

If you inadvertently create an infinite loop with a GOTO statement, halt BASIC with the CTRL/C command.

1.7.2 Conditional Transfer -- IF GOTO and IF THEN

IF GOTO and IF THEN statements use relational operators to test for a specified relationship between two variables, numbers, strings, or expressions. When the relational expression is true, BASIC executes the GOTO instruction. When the IF statement is false, BASIC proceeds to the next line in the program.

The format of the IF GOTO (or IF THEN) statement is

```
(line number) IF v1 relation v2 GOTO x
```

where

v1 and v2	represent variable names, numbers, strings, or expressions
relation	is any relational operator
x	is the number of the line to which BASIC will jump if the relation is true

This example

```
10 LET A=5
20 IF A=2 GOTO 99
30 PRINT "NO"
99 END
```

will cause BASIC to display

NO

BASIC compares strings one alphanumeric character at a time, using ASCII code numbers to determine if one character is "greater" or "less" than another. BASIC proceeds from left to right until it reaches the ends of the strings or until it finds an inequality. If one string is longer than the other, BASIC adds spaces to the shorter string until both are the same length. For example, in comparing AB to a four-letter string, BASIC will treat AB as "AB (space) (space)".

1.7.3 Looping -- FOR, STEP, and NEXT

Programs frequently require the repetition of some instruction or sequence of instructions. One way to achieve this is to write out the steps as many times as you wish BASIC to execute them. For example, this program

```
10 PRINT "HOORAY!"
20 PRINT "HOORAY!"
30 PRINT "HOORAY!"
99 END
```

will instruct BASIC to display HOORAY! in three lines on the terminal.

A better way to achieve the same end is to write the PRINT statement once and instruct BASIC to run through it three times. This type of repetition, which requires BASIC to jump backward in the program and retrace its steps, is called looping.

To execute a loop in a program, BASIC must know two things: which statements to repeat, and how many times to repeat them. FOR and STEP statements let you supply this information.

The format is

```
(line number) FOR v=x TO y[STEP z]
```

where

v	is a variable name. It is the index of the loop, increased or decreased each time the loop is executed.
x	is an expression (numerical value, variable name, or mathematical expression) indicating the initial value of the index -- that is, the value of v before the loop is executed the first time.
y	is an expression indicating the terminal value of the index -- the value of v after the last execution of the loop.
STEP z	is an optional statement used to specify the increment. If you omit it, BASIC assumes a STEP value of 1.

For example, this statement

```
15 FOR K=2 TO 20 STEP 2
```

tells BASIC to repeat the loop as long as K is less than or equal to 20. Since K is incremented by 2 after each execution, BASIC will run through the loop 10 times.

The NEXT statement marks the end of a program loop. It occurs only in combination with a FOR statement.

The format of the NEXT statement is

```
(line number) NEXT v
```

where v is the index variable in the FOR statement.

The NEXT statement causes BASIC to add the STEP value to the index (or to add 1 if the FOR statement contains no STEP value) and to check to see if the value of the index exceeds the terminal value. If it does, BASIC falls through the loop and executes the line following the NEXT statement.

To cause BASIC to exit from a loop before the index has reached the terminal value, use an IF-THEN statement. BASIC can reenter only those loops that it has left before completion.

NOTE

Do not attempt to transfer control from a loop to a subroutine located above it in the program. Doing so may cause BASIC to execute the loop a wrong number of times.

The following example shows one way to use a FOR-NEXT loop to produce the same results of the HOORAY! program above.

```
10 FOR K=2 TO 6 STEP 2
15 PRINT "HOORAY!"
20 NEXT K
99 END
```

The FOR statement tells BASIC to repeat the loop as long as K is less than or equal to 6. Since K is incremented by 2 after each execution, BASIC will run through the loop three times.

1.7.3.1 **Nested Loops** - You may place one or more loops within a loop provided that the inner loops are completely contained by the outer and that no overlapping of loops occurs. Placing one loop within another is called nesting. Each nested loop must have its own FOR and NEXT statements and must terminate before the loop that contains it.

The following examples show legal and illegal types of nested loops:

Legal

```
10 FOR A=1 TO 10
20 FOR B=2 TO 20
30 NEXT B
40 NEXT A
```

Legal

```
10 FOR A=1 TO 10
20 FOR B=2 TO 20
30 NEXT B
40 FOR C=3 TO 30
50 FOR D=4 TO 40
60 FOR E=5 TO 50
70 NEXT E
80 NEXT D
90 NEXT C
95 NEXT A
```

Illegal

```
10 FOR M=1 TO 10
20 FOR N=2 TO 20
30 NEXT M
40 NEXT M
```

The following program contains a nested loop:

```
10 PRINT "INNER",,"OUTER"
15 PRINT
20 FOR I=1 TO 2
30 FOR O=1 TO 3
40 PRINT I,,O
50 NEXT O
60 NEXT I
99 END
```

BASIC will execute the loops and display:

INNER	OUTER
1	1
1	2
1	3
2	1
2	2
2	3

Note that each execution of the outer loop causes BASIC to run through the inner loop three times.

1.7.4 Stopping -- END and STOP

Two statements -- END and STOP -- will cause BASIC to terminate the execution of a program and return control to the editor.

The END statement informs the BASIC compiler that it has come to the last line in the program. Every BASIC program must end with an END statement. No program may contain more than one END statement. A STOP statement cannot take the place of the END statement.

The format of the END statement is

(line number) END

which causes BASIC to return to the edit mode, display

READY

and await your next command.

The STOP statement also terminates a running program, but unlike END, it may occur more than once in the same program.

The format of the STOP statement is

(line number) STOP

The following program demonstrates the use of the STOP statement:

```
10 INPUT A
20 READ B
30 IF A=B GOTO 50
40 STOP
50 PRINT "EQUAL"
60 DATA 3
99 END
```

The STOP statement here prevents BASIC from displaying EQUAL when A does not equal B.

1.7.5 Jumping to Subroutines -- GOSUB and RETURN

A subroutine is a sequence of statements that performs some operation required at more than one point in the program. Subroutines are generally placed at the end of the program, usually before any DATA lines and always before the END statement.

Two statements -- GOSUB and RETURN -- cause BASIC to jump to a subroutine, execute it, and jump back to the point in the main program where it left off. GOSUB and RETURN occur only in combination with each other.

The format of the GOSUB statement is

(line number) GOSUB n

where n is the number of the first line in the subroutine.

When BASIC encounters a GOSUB, it records the number of the line immediately following it and jumps to the first line of the subroutine.

The format of the RETURN statement is

(line number) RETURN

The RETURN statement always occupies the last line in the subroutine. RETURN causes BASIC to jump to the line following the last GOSUB statement it has executed.

You may use the control statements described in this chapter to direct BASIC from one line to another within a subroutine or even to a line in another subroutine.

You may also "nest" subroutines -- use one subroutine to call another -- up to ten levels. If you exceed the tenth level, BASIC prints

GS AT LINE y

where y represents the line number where the error occurred.

The following sample program contains two simple subroutines:

```
10 GOSUB 60
20 PRINT "I'M BACK FROM 1"
30 GOSUB 80
40 PRINT "I'M BACK FROM 2"
50 STOP
60 PRINT "SUBROUTINE 1"
70 RETURN
80 PRINT "SUBROUTINE 2"
90 RETURN
99 END
```

The STOP statement prevents BASIC from "falling into" the subroutines and executing them after it has executed the PRINT statement in line 40. The program will produce:

```
SUBROUTINE 1
I'M BACK FROM 1
SUBROUTINE 2
I'M BACK FROM 2
```

1.8 FUNCTIONS

Functions are special subroutines that perform frequently used operations on numbers and strings.

The format of most functions is

NNN(X)

where

NNN	is a three-letter name
(X)	is an argument enclosed in parentheses. The argument may be a number, a variable, an expression, or another function.

Some functions require multiple arguments and take the form

NNN(X,Y,Z)

Most functions compute a value based on the value of the argument or arguments involved. They are said to "return" this value. For example, SQR(Z) returns the square root of Z.

Functions may return either strings or numbers. Functions that return character strings are distinguished from functions that return numbers by the dollar sign ($) appended to their name. For example, the CHR$ function converts an ASCII code number to its equivalent character and returns the character. The ASC function converts a character to its code number.

Unlike conventional subroutines, functions do not require GOSUB and RETURN statements. They produce their results "in place." For example, the following line will assign the variable A a value of 2:

```
10 LET A=SQR(4)
```

1.8.1 Numeric Functions

BASIC provides numeric functions to perform standard mathematical operations. For example, you may find it necessary to find the sine of an angle. You can do this by looking it up in a table of sine values or by using the BASIC SIN function.

BASIC provides the following trigonometric functions:

- Sine function (SIN)
- Cosine function (COS)
- Arctangent function (ATN)

BASIC provides algebraic functions to find:

- the square root of a number (SQR)
- the value of e -- 2.71828 -- raised to any power (EXP)
- the natural logarithm of a number (LOG)
- the integral part of a number (INT)
- the absolute value of a number (ABS)
- a value based on the sign of a number (SGN).

BASIC also includes a function -- RND -- that returns a random number. You can use this function when you are trying to simulate an unpredictable situation with a BASIC program.

1.8.1.1 **Calculating Sine -- SIN** - The BASIC SIN function lets you calculate the sine of an angle specified in radians. The format is

```
SIN(X)
```

where

X — is a number, numeric variable, expression, or another function, representing the size of an angle in radians

For example, this program

```
10 LET P = 3.14159
20 PRINT SIN(30*P/180)
30 END
```

will display:

```
0.5
```

1.8.1.2 **Calculating Cosine -- COS** - The BASIC COS function lets you calculate the cosine of an angle specified in radians. The format is

```
COS(X)
```

where

X	is a number, numeric variable, expression, or another function, representing the size of an angle in radians

Thus, these lines

```
10 PRINT COS(45*3.14159/180)
20 END
```

will display

```
0.707108
```

1.8.1.3 **Calculating the Arctangent -- ATN** - The BASIC ATN function lets you calculate the angle (in radians) whose tangent is given as the argument of the function.

The format is

```
ATN(X)
```

where

X	is a number, variable, expression, or another function representing the tangent of an angle

Thus, this two-line program

```
10 PRINT ATN(.57735)
20 END
```

will display

```
0.523598
```

1.8.1.4 **Calculating the Tangent** - Although BASIC does not provide a tangent function, you can find the tangent of an angle with the following trigonometric equation:

$$\text{tangent (angle)} = \frac{\text{sine (angle)}}{\text{cos (angle)}}$$

Translated into BASIC, this equation will read

```
10 T=SIN(R)/COS(R)
```

where T is the tangent and R is an angle expressed in radians.

1.8.1.5 **Finding the Square Root -- SQR** - The BASIC SQR function computes the positive square root of an expression. The format is

```
SQR(X)
```

where

X	is a number, variable, expression, or another function

If the argument is negative, the absolute value of the number is used. For example, this program

```
10 PRINT SQR(16)
20 PRINT SQR(-4)
30 END
```

will display

```
 4
 2
```

1.8.1.6 **The Exponential Function -- EXP** - The BASIC EXP function calculates the value of e raised to the X power, where e is equal to 2.71828. That is, EXP(X) is equivalent to 2.71828**X.

The format is

```
EXP(X)
```

where

X	is a number, numeric variable, expression, or another function

Thus, this program

```
10 PRINT EXP(1.5)
20 END
```

will display

```
 4.48169
```

1.8.1.7 **Calculating the Natural Logarithm -- LOG** - The BASIC LOG function calculates the natural logarithm of X (to the base e).

The format is

```
LOG(X)
```

where

X	is a number, numeric variable, expression, or another function

EXP and LOG perform opposite functions. That is the exponent x (the input in the EXP function) in the formula e^x=y is the logarithm of y to the base e (the output of the LOG function) in the formula x=log(e) y.

This BASIC formula demonstrates their relationship:

```
LOG(EXP(X)) = X
```

1.8.1.8 **The Integer Function -- INT** - The BASIC INT function returns the value of the largest integer not greater than the argument. The format is

```
INT(X)
```

where

X	is a number, numeric variable, expression, or another function

To round off a number to the nearest integer, specify INT(X+.5).

For example, this function

```
10 INT(34.67)
```

returns the value 34;

these functions

```
10 INT (34.67 + .5)
15 INT (34.36 + .5)
```

return the values 35 and 34;

this function

```
10 INT (-14.37)
```

returns the value -15.

1.8.1.9 **The Absolute Value Function -- ABS** - The BASIC ABS function returns the absolute value of an expression. The format is

```
ABS(X)
```

where

X is a number, numeric variable, or numeric expression

By mathematical definition, the absolute value of a number which represents its magnitude is always positive. The absolute value of a positive number is equal to the number; the absolute value of a negative number is equal to the number times -1. For example, this program:

```
10 PRINT ABS(-5)
20 END
```

will display

```
5
```

1.8.1.10 **The Sign Function -- SGN** - The SGN function lets you determine if an expression is positive, negative, or equal to zero. The format is

```
SGN(X)
```

where

X is a number, numeric variable, expression, or another function

If the argument is any positive number, the SGN function will return a value of 1. If the argument is negative, SGN returns -1. If it is 0, SGN returns 0. For example, these lines

```
10 LET A=5\LET B=0\LET C=-2
20 PRINT SGN(A);  SGN(B);  SGN(C)
```

will display

```
1  0  -1
```

because

5>0,
0=0, and
-2<0.

1.8.1.11 **Random Numbers -- RND** - A random-number series is a series of numbers that are not related to each other in any way. You can use random numbers in a BASIC program to simulate a situation in which the outcome is not predictable -- the flip of a coin, for example, or the rolling of dice.

It is not possible to produce a series of truly random numbers on a computer since, given the same starting conditions, a computer always comes up with the same results. Instead, BASIC uses complex calculations to generate a series of numbers that seem unrelated. This is called a pseudo-random series.

The BASIC RND function produces pseudo-random numbers between -- but not including -- 0 and 1. The format is

```
RND(X)
```

where

X is a dummy variable. Type the function just as it appears above.

Each time BASIC encounters the RND function in a program, it produces a different decimal number. However, if you run the program again, BASIC will output the same set of numbers. To generate a different set of numbers with each execution, use the RANDOMIZE statement in your program.

The format of the RANDOMIZE statement is

```
(line number) RANDOMIZE
```

The following routine will print a different series of random numbers each time you run it. (RANDOMIZE uses the value you enter to vary the output.)

```
10 INPUT X
20 FOR L=1 TO 20
30 PRINT INT (10*RND(X));
40 NEXT L
50 IF X <0 GOTO 20
99 END
```

1.8.2 String Functions

BASIC string functions let you examine and modify strings and perform certain conversions between numbers and strings. Functions that return strings are distinguished from functions that return numbers by the dollar sign ($) after their name.

BASIC provides three functions that allow you to analyze and manipulate strings:

- LEN function -- determines the length of a string
- POS function -- searches for the position of a set of characters within a string
- SEG$ function -- copies a segment from a string

Other functions enable you to convert strings to numbers and numbers to strings:

- ASC function -- converts a character to its ASCII code equivalent
- CHR$ -- converts an ASCII code number to a character
- STR$ -- converts a number to its string representation
- VAL -- converts a string representation of a number to a number

1.8.2.1 **Finding the Length of a String -- LEN** - The LEN function returns the number of characters in a string.

The format is

```
LEN(X$)
```

where

X$ is a string, a string variable, or several concatenated strings and/or string variables

For example,

(1) This line:

```
10 PRINT LEN("DOG")
```

will display

```
3
```

(2) This program:

```
10 A$="UP, "
20 B$="DOWN, AND "
30 PRINT LEN(A$&B$&"AROUND")
40 END
```

will display

```
20
```

because

```
"UP," =          4 characters
"DOWN, AND" =   10 characters
"AROUND" =       6 characters

Total           20 characters
```

If a string has never been defined, it will have a length of 0.

This program:

```
20 PRINT LEN(L$)
99 END
```

will display

```
0
```

1.8.2.2 **Finding a Substring -- POS** - The BASIC POS function returns the location of a specified group of characters in a string.

The format is

```
POS(X$,Y$,Z)
```

where

X$	is the string you want to search
Y$	is the substring you are searching for
Z	is the position in the string at which you want to begin the search

This function searches X$ for the first occurrence of Y$. It begins the search with the Zth character in X$. Depending on what it finds, POS returns the following results.

1. If it finds substring Y$, POS returns the position of the first character in the series.
2. If it fails to find Y$, POS returns a 0.
3. If Y$ is a null string (containing no characters), POS returns a 1.
4. If X$ is a null string, POS returns a 0.

NOTE

If Z is less than 0 or greater than the string, BASIC prints an error message and stops the program.

These lines cause the POS function to start at the seventh character in the string "ABCDEFGHIDEF" and search for the substring "DEF":

```
10 DIM B2$(12)
20 B2$ = "ABCDEFGHIDEF"
30 PRINT POS (B2$, "DEF", 7)
```

POS returns 10. (Change the 7 to a 1 in line 30 and POS will return a 4.)

1.8.2.3 **Displaying a Substring -- SEG$** - The SEG$ function searches for a segment -- a substring -- of a string and returns it for display.

The format is

```
SEG$(X$,Y,Z)
```

where

X$	is the string containing the substring you want to display. X$ may be a variable or the string itself.
Y	is the position of the first character in the substring
Z	is the position of the last character in the substring

SEG$ returns a null string (no characters) if:

- Y is greater than the length of X
- Z is less than 1
- Z is less than Y

If Y is less than 1, SEG$ sets it to 1. If Z is greater than the length of X$, SEG$ sets it equal to the length of X$.

These lines

```
10 DIM B2$(12)
20 B2$ = "ABCDEFGHIDEF"
30 PRINT SEG$(B2$,3,5)
```

will display:

```
CDE
```

1.8.2.4 **Converting a Character to ASCII Code -- ASC** - The ASC function converts a one-character string to its ASCII code equivalent. The format is

```
ASC(X)
```

where

X	is a one-character string

ASC returns the equivalent decimal number for the argument. Table 1-1 lists all the alphanumeric characters available on the terminal and their ASCII code numbers.

Table 1-1
Alphanumeric Characters and Corresponding ASCII Code Numbers

Character	Decimal	Character	Decimal
@	0	(space)	32
A	1	!	33
B	2	"	34
C	3	#	35
D	4	$	36
E	5	%	37
F	6	&	38
G	7	'	39
H	8	(	40
I	9	)	41
J	10	*	42
K	11	+	43
L	12	'	44
M	13	-	45
N	14	.	46
O	15	/	47
P	16	0	48
Q	17	1	49
R	18	2	50
S	19	3	51
T	20	4	52
U	21	5	53
V	22	6	54
W	23	7	55
X	24	8	56
Y	25	9	57
Z	26	:	58
[	27	;	59
\	28	<	60
]	29	=	61
^	30	>	62
_	31	?	63

Thus, this program

```
10 LET A$="*"
20 PRINT ASC("P"),ASC(A$),ASC(")
30 END
```

will display

```
 16          42          57
```

1.8.2.5 **Converting ASCII Code to a Character -- CHR$** - The CHR$ function converts a code number to its equivalent character.

The format is

```
CHR$(X)
```

where

X is a number, a numeric expression, or a numeric variable

CHR$ returns the equivalent character for the argument. (See the ASC function for the table of decimal/character conversions.)

If the argument is greater than 63, divide it by 64 and use the remainder to search the table.

Thus, this line:

```
10 PRINT CHR$(1),CHR$(40)
```

will display

```
A       (
```

This line:

```
10 PRINT CHR$(207),CHR$(77)
```

will display

```
O       M
```

207/64 = 3, with a remainder of 15
 77/64 = 1, with a remainder of 13

Using 15 and 13 to search the table yields the letters "O" and "M".

1.8.2.6 **Converting Numbers from String to Numeric Format -- VAL** - The VAL function converts numbers in string form to numeric data. The format is

```
VAL(X$)
```

where

X$ is a string made up of those values that BASIC accepts as numeric data. These are:

- digits 0 through 9
- + or - sign
- the letter E
- leading spaces -- BASIC ignores them
- the first decimal point (.)

Keep in mind that BASIC does not consider numbers and numeric expressions in string form as numeric data. It will not use them in calculations or as arguments in mathematical functions until you convert them into numeric format with the VAL function.

This program instructs BASIC to read a string, convert it into numeric form, and multiply it by two:

```
10 INPUT A$;
20 PRINT VAL(A$)*2
30 END
```

BASIC displays:

```
?2.46111
 4.92222
```

1.8.2.7 **Converting a Number to a String -- STR$** - The STR$ function converts numerics to strings. The format is

```
STR$(X)
```

where

X is a numeric expression

The STR$ function returns the string value of the expression exactly as BASIC would print it but without a leading or trailing space. Use the STR$ function when you want to print a number without a leading or trailing space and when you want to perform string operations or functions on a number.

1.8.3 User-Defined Functions

1.8.3.1 **The FNa Function and the DEF Statement** - In some programs, you may want to perform the same sequence of string or numeric operations more than once. As an aid in such cases, BASIC lets you define your sequence as a special function -- called a user-defined function -- that you can call for in the same way you would call for any string or numeric function that BASIC provides.

The BASIC DEF statement lets you create user-defined functions. The format of the DEF statement is

```
(line number) DEF FNa (list) = expression
```

where

(list) contains the dummy variable or variables that appear in your operation. The same variables must appear in the expression.

expression is the operation you want BASIC to perform each time you call for the function. The operation may contain numbers, several variables, other functions, or mathematical expressions.

For example, if you write a program in which you repeatedly use the operation e^-x2+5, you can introduce it as a user-defined function with this DEF statement:

```
30 DEF FNE(X)=EXP(-X**2)+5
```

and then call for various values of the function -- FNE(.1), FNE(3.45), FNE(A+2), etc.

This statement:

```
10 DEF FNA(S)=S**2
```

will cause the user-defined function in this line

```
20 LET R=FNA(4)
```

to return a 16.

If the function involves more than one variable, BASIC will identify them by their position. For example, this program

```
10 DEF FNH(N,P)=2*P+N
20 LET X=4\LET Y=5
30 PRINT FNH(X,Y)
40 END
```

will display

```
14
```

BASIC takes the first value in the function (4) as "N", because "N" appears first in the DEF statement. It takes the second value (5) as "P", because "P" is in the second position.

```
DEF FNH (N,P) = 2*P+N


first position          second position


PRINT FNH(X,Y)
```

You must introduce each user-defined function with a separate DEF statement, taking care to place each DEF statement before the first occurrence of the function it defines. For example, if you want to use a special function called FNB(X) in your program, you must first write a DEF statement with FNB as the parameter. You may define up to 26 FN functions in the same program (FNA, FNB...,FNZ).

1.8.3.2 **The UDEF Function Call and the USE Statement** - OS/8 BASIC lets you add one or more user-coded assembly-language functions to a BASIC program and use them in the same way you would use any other function. For complete instructions to write and interface such functions, see Chapter 2.

To specify a user-coded function in an OS/8 BASIC program, type

```
line number UDEF function name(a,b,c)
```

where

function name	consists of alphabetic characters only and has at least one argument (a dummy, if necessary)
(a,b,c)	are arguments. User-written assembly-language functions may contain up to four numeric and two string arguments.

For example:

```
10 LET R=4
15 LET B=6
20 LET Q=10
25 UDEF PLT(X,Y,Z)
30 LET D=PLT(R,B,0)
35 PRINT 4*D
40 END
```

Line 25 introduces the function PLT to OS/8 BASIC and indicates the number and type of arguments associated with it. In line 30 the function appears as any standard function might appear in a BASIC program. If the function requires an array, a USE statement identifying the array must precede the statement that calls the function. Thus:

```
10 DIM S(15,5)
 .
 .
20 LET Q=10
22 USE S
25 UDEF PLT(X,Y,Z)
 .
 .
 .
```

1.8.4 The Debugging Function -- TRC

The TRC function causes BASIC to print the line numbers of statements in a program in the order that it executes them. This lets you follow the course of loops and subroutines and provides a useful tool for debugging a program.

The format of the TRC function is

```
v = TRC(X)
```

where

v	is any letter. It has no purpose except to occupy the position in the line.
X	is 1 or 0. 1 turns the function on; 0 turns it off.

When it comes upon a TRC(1) in a program, BASIC begins displaying the line number (enclosed by percent signs) of each statement it executes -- with the exception of the following types: DATA, DEF, DIM, END, GOTO, NEXT, RANDOMIZE, REM, and STOP. Encountering a TRC(0) will cause it to stop outputting line numbers and resume normal operation.

For example, this program:

```
60 T=TRC(1)
70 GOSUB 90
80 GOTO 140
90 PRINT "IN OUTER SUB"
100 GOSUB 120
110 RETURN
120 PRINT "IN INNER SUB"
130 RETURN
140 T=TRC(0)
150 END
```

will display

```
% 70 %
% 90 %
IN OUTER SUB
% 100 %
% 120 %
IN INNER SUB
% 130 %
% 110 %
% 140 %
```

1.8.5 Calling for the Date -- the DAT$ Function

The DAT$ function returns the current system date.

The format is

```
DAT$(X)
```

where

X is a dummy variable

Enter this function exactly as it appears above. DAT$ returns an eight-character string in the form

```
mm/dd/yy
```

For example, these lines:

```
10 LET D$ = DAT$(X)
20 PRINT D$
```

will display

```
07/20/77
```

if that date was entered with the monitor DATE command.

If you have not specified the date with the MONITOR date command, the function will return no characters.

1.9 FILE STATEMENTS

BASIC file statements -- which are distinguished from other BASIC statements by the number sign (#) -- let you store data on peripheral devices for later use in any BASIC program. They include:

- FILE# Describes the file, assigns it a channel number from 1 to 4 (the number of files that BASIC can handle at one time), and opens it.
- INPUT# Reads data from the file.
- PRINT# Writes data on the file.
- RESTORE# Resets the pointer to the beginning of the file.
- CLOSE# Closes the file and removes the channel number.
- IF END# Tests for end-of-file.

In most operations, you open a file (FILE#) for input or output, read from it (INPUT#) or write on it (PRINT#), and close it (CLOSE#). You may open only four files at a time -- excluding the terminal, which is always open and available for use. However, the ability to open and close files under program control gives you access to an unlimited number of files. That is, when you close a file, you may reassign its channel number to a newly opened file.

BASIC treats files in the same way it treats terminal input and output. The INPUT statement causes BASIC to read a value that you enter on the terminal and assign it to a variable; the INPUT# statement causes it to read a value from a file. The PRINT statement instructs BASIC to display data on the terminal; the PRINT# statement tells it to write data in a file.

BASIC uses two types of file: string files and numeric files. You may write numbers into a string file in both string and numeric format. Numeric files, however, may contain numeric data only.

1.9.1 File Control

You must open a file with a FILE# statement before you can read or write any data. You should close any files that you open during the course of a program with a CLOSE# statement. The CLOSE# statement cancels the channel number that you have assigned with the FILE# statement, making the channel available to any other newly opened file.

1.9.1.1 **Opening a File -- FILE#** - The BASIC FILE# statement opens a file for input or output, defines it, and assigns a channel number. An input file is one you are reading from. An output file is one you are writing to.

The format of the FILE# statement is

(line number) FILE t#n:"filespec"

where

t	is one of the following: (blank) for an input string file V for an output string file N for an input numeric file VN for an output numeric file
n	is the channel number (1 through 4) that you are assigning to the file. It can be a numeric variable.
"filespec"	is an OS/8 device, file name, and extension. It must either be a string enclosed by quotation marks or a string variable.

You must include a channel number (n) in all FILE# statements. (The channel number of the terminal is always FILE#0.)

For example, this statement describes the string file RXA1:DATA2.AS as file number 1 and opens it for output:

```
10 FILEV#1:  "RXA1:DATA2.AS"
```

This statement describes the numeric file MONEY.NU on RXA1 as file number 2 and opens it for output:

```
10 FILEVN 2:  "RXA1:MONEY.NU"
```

These statements describe the string file RXA1:TEST.AB as file number 3 and open it for input:

```
10 LET A$="RXA1:TEST.AB"
15 FILE#3:  A$
```

This statement describes the numeric file RXA1:FIL3.CD as file number 4 and opens it for input:

```
10 FILEN#4: "RXA1:FIL3.CD"
```

1.9.1.2 **Closing a File -- CLOSE#** - The CLOSE# statement closes any file you specify and disassociates it from its channel number. This allows BASIC to reassign the number to another file. After you close a file, you cannot use it again until you reopen it.

The format of the CLOSE# statement is

(line number) CLOSE# n

where

n	is the channel number of the file to be closed (or a variable)

You must close all output files in a program before instructing BASIC to execute an END, STOP, or CHAIN statement. If you do not close them, they will be lost.

In the following program, the CLOSE# statement allows BASIC to reassign the channel number of file SYS:TEST.XX to the newly opened file RXA1:FILD:DA:

```
50 FILEV #1:"SYS:TEST.XX"
60 PRINT #1:"A","B","C","D"
70 CLOSE #1
80 FILE #1:"RXA1:FILD.DA"
90 INPUT #1:J$
```

1.9.2 File I/O

You use BASIC files in the same way you use the terminal for sequential input and output. The difference is that files allow you to manipulate much more data in much less time than the terminal.

You can open a file to supply input or to receive output, but you cannot open it to do both at the same time. To update an existing file, you must open it for input, open a new file for output, read the data from the input file and write the data including any changes you wish to make on the output file.

1.9.2.1 **Reading Data from a File -- INPUT#** - The INPUT# statement instructs BASIC to read data from a file and assign values to specified variables. BASIC reads file data serially. This means that it must read through an entire list to get at the last item of data.

The format of the INPUT# statement is

(line number) INPUT#n:variables

where

n	is the channel number of the file you are reading (or a variable)
variables	is the list of variables -- separated by commas -- into which BASIC will read data

The INPUT# statement automatically steps through the file item by item to find values to satisfy its variables.

In most operations, you write numbers into numeric files and strings into string files and then read them back into the corresponding variables. If you wish, however, you may write numbers into string files and read them back into either numeric or string variables, depending on how you want to use them. If you assign numbers from a string file to string variables, they will appear in string form and be subject to the same rules as other strings. If you assign numbers from a string file to numeric variables, BASIC will convert them into numeric form. Keep in mind that string files contain carriage returns and line feeds. These will appear as zeros if read into numeric variables.

For example, the following program instructs BASIC to write numbers into a string file and read them back as numeric data. The "C" and "L" variables in the INPUT# statement in line 80 receive the zeros generated by the carriage return and the line feed.

```
10 FILEV#1:"SYS:FILA.ZZ"
20 FORI#1 TO 5
30 PRINT #1:I
40 NEXT I
50 CLOSE#1
60 FILE#1:"SYS:FILA.ZZ"
70 FOR I=1 TO 5
80 INPUT#1:J,C,L
90 PRINT J
100 NEXT I
110 END
```

It will display

```
1
2
3
4
5
```

1.9.2.2 **Writing Data on a File -- PRINT#** - The PRINT# statement lets you write data on an output file. Its format is

(line number) PRINT# n: expression

where

n — is the channel number or a variable representing the channel number

expressions — may be numerics or strings, depending on the type of output file you have opened in the FILE# statement

- If you open a string output file (FILEV#), the expressions may be string or numeric, separated by commas or semicolons. You may use the TAB and PNT functions when writing on string files. (See Section 1.6.4.3.)

- If you open a numeric output file (FILEVN#), the expressions must be numbers or numeric variables, separated by commas or semicolons.

When you use the PRINT# statement to write data into an output string file, BASIC interprets commas, semicolons, and RETURNs the same way it interprets them in PRINT statements. For example,

```
10 PRINT "A", "B"
20 PRINT "C"; "D";
30 PRINT "E"
```

will display

```
A       B
CDE
```

The following lines will cause the same display:

```
5 DIM J$(30)
10 FILEV#2:"RXA1:PROG.XX"
20 PRINT #2:"A", "B"
30 PRINT #2:"C"; "D";
40 PRINT #2:"E"
50 CLOSE#2
60 FILE#2:"RXA1:PROG.XX"
70 INPUT#2:J$
80 PRINT J$
90 INPUT#2:J$
100 PRINT J$
```

When you use the PRINT# statement to write data into an output numeric file, BASIC converts commas and semicolons to spaces. The file will simply contain a "list" of numbers separated by spaces. For example, this program

```
10 FILEVN#1:"SYS:TST.XX"
20 PRINT#1:1,2
30 PRINT#1:3,4,
40 PRINT#1:5,6
50 CLOSE#1
60 FILEN#1:"SYS:TST.XX"
70 FOR X=1 TO 6
80 INPUT#1:Z
90 PRINT Z
100 NEXT X
999 END
```

will display

```
 1
 2
 3
 4
 5
 6
```

1.9.2.3 **Resetting a File -- RESTORE#** - The RESTORE# statement resets the file back to the beginning so that the next INPUT# statement will cause BASIC to read the first item in the series. The format is

(line number) RESTORE# n

where

n — is the channel number of the file to be reset or a variable representing the channel number

If n is 0, BASIC resets the DATA list to the beginning.

In the following program, RXA1:FILB.LM is a numeric input file containing the numbers 1 through 9. These instructions:

```
100 FILEN#3:"RXA1:FILB.LM"
110 FOR I=1 TO 3
120 INPUT #3:Z
130 PRINT Z
140 NEXT I
150 RESTORE#3
160 INPUT#3:Z
170 PRINT Z
999 END
```

will display

```
1
2
3
1
```

1.9.2.4 **Checking for End-of-File -- the IF END# Statement** - The IF END# statement lets you detect the end of a string file. The format is

(line number) IF END# n THEN m

where

n	is the channel number of the file in question or a variable representing the number
m	is the number of the line in the program to which BASIC will jump if it has reached the end of the file

The IF END# statement works only on string files and must immediately follow the PRINT# or INPUT# statement for that file.

When you use the IF END# statement, you are asking BASIC to check if its last attempt to execute a PRINT# or INPUT# statement was successful. If it was unsuccessful -- if nothing was written or read -- BASIC jumps to line m.

For example, in this program

```
10 FILEV#1: "SYS:PROGA.BB"
20 PRINT#1: "A"
30 PRINT#1: "B"
40 CLOSE# 1
50 FILE#1: "SYS:PROGA.BB"
60 INPUT#1:A$
70 IF END#1 THEN 100
80 PRINT A$
90 GOTO 60
100 PRINT "END OF FILE"
110 CLOSE#1
```

the lines will be executed in this sequence

```
10
20
30
40
50
60
70
80
90
60
70
80
90
60
70
100
110
```

so that the display will be

```
A
B
END OF FILE
```

1.10 SEGMENTING PROGRAMS -- THE CHAIN STATEMENT

FILE# statements let you manipulate data files under program control. The CHAIN statement (used in connection with the SAVE command) lets you do the same thing with files that contain programs.

With the SAVE command, you can divide a long program into shorter segments and then store the pieces in separate files. During program execution, CHAIN statements cause BASIC to retrieve the segments one after another and run them together in a chain.

The format of the CHAIN statement is

```
CHAIN "filespec"
```

where

"filespec"	is the device and file name -- enclosed by quotation marks -- of the program you want to run

When BASIC encounters a CHAIN statement in a program, it stops execution to retrieve, compile (if necessary), and run the program you have called for. After BASIC has run all the programs in the chain, the workspace and the BASIC.WS file will both contain the program it started with.

Since BASIC removes each program from core memory before retrieving the next one in the chain, you must be sure to CLOSE# all data files in any program containing a CHAIN statement. If you do not, data will be lost.

Programs for chaining must all be the same type. A BASIC source program will chain only to another BASIC source program, and a memory image file (identified by the .SV extension in the file name) to another memory image file.

NOTE

When chaining BASIC memory image files, you must place the program being chained to on SYS. This is a restriction of the USR CHAIN function.

In the following example, during a run of program PROG1.BA, the CHAIN statement causes BASIC to halt execution to retrieve and execute the program called CHAIN1.BA. The CHAIN statement in this program, in turn, causes CHAIN2.BA to run, completing the series.

```
NEW CHAIN1.BA
READY

10 PRINT "FIRST LINK"
20 CHAIN "SYS:CHAIN2.BA"
99 END

SAVE SYS: CHAIN1.BA

READY

NEW CHAIN2.BA

10 PRINT "SECOND LINK"
99 END

SAVE SYS: CHAIN2.BA

READY

NEW PROG1.BA

10 PRINT "CHAIN STARTS HERE"
20 CHAIN "SYS:CHAIN1.BA"
99 END

RUNNH

CHAIN STARTS HERE
FIRST LINK
SECOND LINK
```

In general, any departure from these procedures will produce a CX error.

1.11 BASIC COMMANDS

BASIC commands let you create, modify, store, and run programs under the direction of the BASIC editor. To summon the editor, type BASIC in response to the OS/80 monitor dot. The editor will respond with the message

```
NEW OR OLD --
```

indicating that it has assumed control of the system and reserved a special area in memory -- called the workspace -- for your program. You may now tell BASIC whether you wish to enter a new program or call for one that you have previously written and stored on a peripheral device.

1.11.1 Entering a New Program -- the NEW Command

The NEW command clears the workspace and tells the editor the name of the program you are about to enter.

The format is

```
NEW filename[.ex]
```

where

filename.ex	is the name and extension of the new program you are about to enter. If the extension is omitted, BASIC calls it ".BA".

If you strike the RETURN key immediately after typing NEW, BASIC clears the workspace and prompts with the message

```
FILE NAME --
```

You must now type the file name and extension and press the RETURN key.

Thus, the following commands both instruct BASIC to clear the workspace and name a new program "TEST.BA":

```
NEW TEST
NEW TEST.BA
```

You enter a BASIC program line by line, keeping in mind that you must:

- begin each line with a number. Line numbers may range from 1 to 99999 and must contain no internal spaces or nonnumeric characters.
- terminate each line with the RETURN key.

If you make a typing error, you may correct it by striking the DELETE key once for each error you wish to erase. If you wish to delete the entire line, press the CTRL/U key command.

1.11.2 Calling for an Old Program -- the OLD Command

The OLD command instructs BASIC to clear the workspace, find a file on a peripheral device, and place it in the workspace. The format is

```
OLD dev:filename[.ex]
```

where

dev:filename.ex	is the device, file name, and extension of the program you are calling for. If you omit the extension, BASIC assumes ".BA".

If you strike the RETURN key immediately after typing OLD, BASIC clears the workspace and prompts with the message

```
FILE NAME --
```

You must now type the file name and extension and press the RETURN key.

These two commands both cause BASIC to clear the workspace and bring TEST.BA into the workspace from RXA1:

```
OLD RXA1:  TEST.BA
OL RXA1:  TEST
```

1.11.3 Running a Program -- the RUN Command

The RUN command instructs BASIC to display a header line (containing the file name and extension, BASIC version number, and the date) and execute the program in the workspace. The RUNNH command causes it to run the program without the header.

The format is

```
RUN
```

or

```
RUNNH
```

To run a program, BASIC first reserves space in memory for all arrays dimensioned in DIM statements, defines user functions in DEF statements, and initializes all numeric variables at zero and all string variables at null string. Then it begins execution at the lowest line number.

If BASIC encounters no errors, it will complete execution and display any data you asked for in PRINT statements. When it has finished, it will signal

```
READY
```

NOTE

The RUN and RUNNH commands also cause BASIC to store a copy of the program it is running in a file called BASIC.WS.

If you neglect to save the program with a SAVE command or if for some reason you cannot retrieve it, call for OLD file BASIC.WS. Keep in mind that the program in BASIC.WS is always the last one you have run.

1.11.4 Displaying a Program -- the LIST Command

The LIST command causes BASIC to print a header line (containing the file name and extension, BASIC version number, and date) and display the program currently in the workspace. The LISTNH command instructs BASIC to suppress the header.

The format is

```
LIST [n]
```

or

```
LISTNH [n]
```

where

n	is a line number in the program

If n is present, the LIST command will cause BASIC to display the line number n and all the lines following it in the program. If n is omitted, BASIC will display the entire program.

To terminate a listing, type the CTRL/O key command.

Use the LIST command when correcting or modifying the program in the workspace. For example, if BASIC informs you that an error exists in line 30, type LIST 30 to see the line.

```
30 IF A=X GOTO
```

When you have detected the error -- in this case the omission of a line number after GOTO -- rewrite the entire line correctly and press the RETURN key.

1.11.5 **Storing a Program -- the SAVE Command**

The SAVE command causes BASIC to take the file currently in the workspace and store it on any device you specify.

The format is

```
SAVE [dev:filename.ex]
```

where

dev:filename.ex	is the device, file name, and extension of the program you want to store. If you omit the device, BASIC stores the file on DSK:. If you omit the file name, BASIC uses the name you gave it in a NEW or OLD command.

The SAVE command provides you with a way to list large programs on the line printer rather than the terminal. Type

```
SAVE LPT:
```

to list the contents of the workspace on the line printer.

1.11.6 Renaming a Program -- the NAME Command

The NAME command lets you rename the file currently in the workspace.

The format is

```
NAME filename.ex
```

where

```
filename.ex    is the new name of the program
```

Since this command changes only the name of the file in the workspace -- not the file itself -- you can use it to create and save two similar versions of the same program. To do this:

1. Read the program into the workspace with the OLD command.
2. Rename the contents of the workspace.
3. Make the changes.
4. Save the new version under the new name.

1.11.7 Erasing the Workspace -- the SCRATCH Command

The SCRATCH command tells BASIC to erase everything from the workspace, leaving you a clean area in which to write.

The format is

```
SCRatch
```

The OLD and NEW commands also clean the workspace. Nevertheless, it is good programming practice to use the SCRATCH command before entering a new program or calling for an old one.

1.11.8 Leaving Basic -- the BYE Command

The BYE command dismisses the BASIC editor and returns control of the system to the OS/8 monitor.

The format is

```
BYE
```

Never give the BYE command without first saving the program in the workspace. When you call BASIC again and respond to the NEW or OLD message, BASIC will erase the workspace, destroying the program.

1.11.9 Resequencing a Program -- Calling RESEQ

After you have made extensive modifications in a program, you may find that some parts now contain consecutively numbered lines, making it difficult to insert additional statements where you may need them. The BASIC RESEQ program renumbers your program and lets you specify a suitable increment between lines. RESEQ automatically changes the line numbers in GOSUB and IF THEN statements to agree with the renumbered program.

Programs for RESEQuencing must not exceed 350 lines. The lines must not exceed 80 characters.

Here is an example of a typical resequencing operation:

Command	Meaning
SAVE DSK:SAMPLE.BA	You save SAMPLE (which is the program you want to resequence).
READY	BASIC indicates it is ready to receive your next command.
OLD DSK:RESEQ	You call for program RESEQ.
READY	BASIC is ready for your next command.
RUNNH	You tell BASIC to run RESEQ.
FILE? DSK:SAMPLE.BA	RESEQ program asks for file name. You respond with device, name, and extension of program you want to renumber.
START,STEP? 100,10	RESEQ asks for a starting line number (START) and an increment between line numbers (STEP). You specify a starting number of 100 and an increment of 10.
READY	When RESEQ has finished renumbering your program, BASIC indicates that it is ready for your next command.
OLD DSK:SAMPLE.BA	You call back your program.
READY	BASIC is ready for your next command.
LISTNH	You tell BASIC to display program SAMPLE on terminal.

Don't worry if renumbering seems slow. This is a characteristic of the RESEQ program.

1.11.10 Key Commands

BASIC key commands let you delete characters and lines that you have typed, interrupt execution of BASIC programs, and control listings on the terminal. To type a CTRL command, hold down the CTRL key and press the appropriate letter.

1.11.10.1 **Correcting Typing and Format Errors -- DELETE, CTRL/U** - To correct typing errors, press the DELETE key. Each time you strike the key, another character is deleted.

Sometimes you may find it easier to delete an entire line rather than making corrections with a series of DELETEs. To erase an entire line, type CTRL/U. This key command -- which is equivalent to typing DELETE back to the beginning of the line -- erases the line, echoes "DELETED", and performs a line feed.

1.11.10.2 **Eliminating Program Lines -- RETURN** - To delete a line from a BASIC program, type the line number and press the RETURN key. This removes both the statement and the line number from the program.

1.11.10.3 **Interrupting Program Execution -- CTRL/C** - To stop a program during execution, type CTRL/C. BASIC responds with READY, allowing you to correct or modify the program.

NOTE

If you type CTRL/C after the READY message appears, BASIC will return control to the OS/8 monitor.

1.11.10.4 **Controlling Program Listings on the Terminal -- CTRL/S, CTRL/Q, and CTRL/O** - If your program exceeds a single display frame -- 24 lines -- you may wish to stop the scrolling caused by the LIST/LISTNH commands.

The following key commands let you control listings.

CTRL/S	Suspends scrolling in the display frame.
CTRL/Q	Resumes scrolling.
CTRL/O	Causes BASIC to abort listing and signal with READY message.

CHAPTER 2

CREATING ASSEMBLY LANGUAGE FUNCTIONS

2.1 INTRODUCTION

Experienced programmers may write original routines and functions in assembly language and run them with BASIC programs. Such operations require knowledge of the BASIC run-time system (BRTS), since BRTS is the part of BASIC that executes all user-written programs, functions, and routines. The following chapter, which includes a detailed description of BRTS, assumes that the reader is familiar with the OS/80 assembly language PAL8.

BASIC consists of five discrete parts:

1. The BASIC editor, which enables you to create and edit source programs. When you type a RUN command, the editor opens a temporary file called BASIC.WS, stores the source program in the file, and chains to the compiler.

2. The BASIC compiler, which translates the source program into a pseudo-code.

3. The loader, which places the pseudo-code into memory along with the run-time system.

4. The BASIC run-time system, which interprets pseudo-code and calls overlays into core memory as it needs them.

5. The BRTS overlays, which consist mainly of BASIC functions. BRTS reserves one of these for user-written assembly-language functions and subroutines.

The following chart lists the names of the files in the BASIC system and the file names of the programs each produces or uses during run time.

BASIC Component	File Name	Associated File	Use
Editor	BASIC.SV	BASIC.WS	source program storage
Compiler	BCOMP.SV	BASIC.WS	source program storage
		BASIC.TM	compiled code storage
Loader	BLOAD.SV	BASIC.TM	compiled code storage
BRTS	BRTS.SV	BASIC.AF	overlays of functions,
		BASIC.SF	if needed
		BASIC.FF	
		BASIC.UF	

Note that these file names identify programs in the BASIC system. You must not use them to identify your own programs.

2.2 THE BASIC RUN-TIME SYSTEM - BRTS

The BASIC run-time system executes all user-written programs, including original assembly-language functions. The description in this chapter of the configuration of BRTS during execution uses the following conventions:

- Memory locations have symbolic names (always capitalized). You may obtain the actual value of these symbols from the symbol table for the version of BASIC you are using.
- The symbol table is for a non-EAE system. If the EAE overlay is used, some minor symbols will change. The major routine entry points, however, are the same in both systems.
- Variable names used in this chapter -- A, A(0,0), A$, and A$(0) -- represent the general case.
- All references to "page 0" indicate BRTS page 0 (page 0, field 0).
- All diagrams in this chapter locate the lowest memory address at the top.

During execution, BRTS has the following configuration in memory.

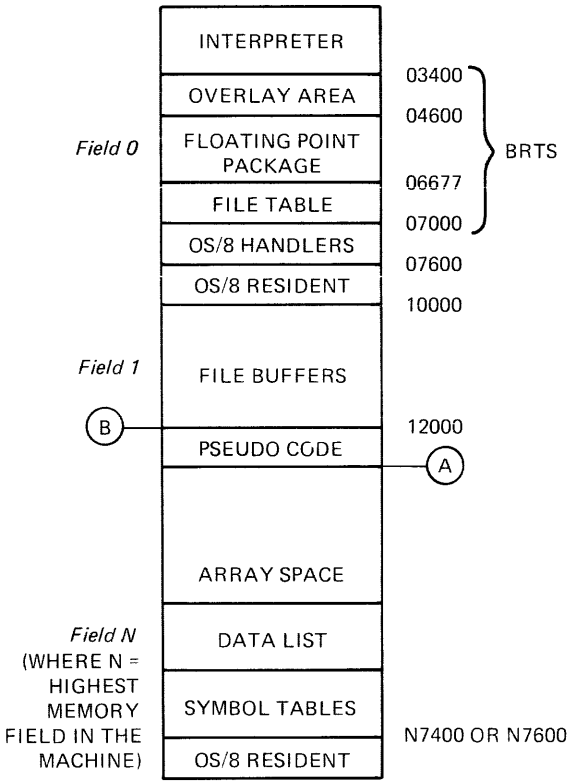


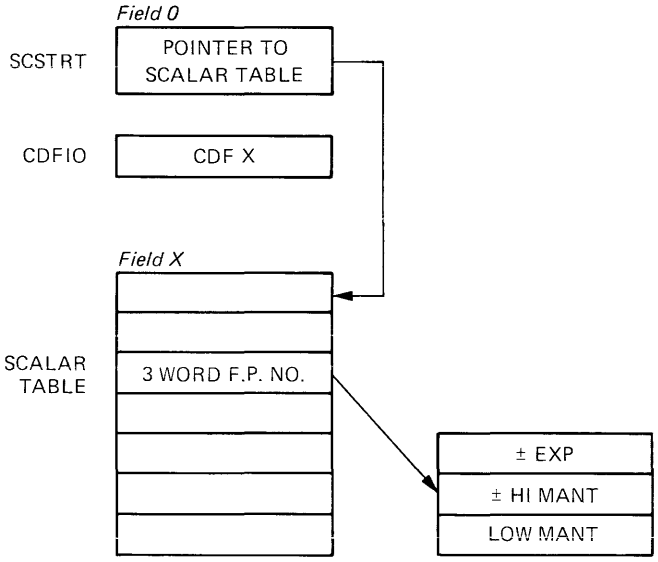
Figure 2-1 BRTS Configuration

2.2.1 BRTS Symbol Tables

BRTS reserves space in the highest field in memory for its four symbol tables, which it uses to locate variables during run time. These tables include the scalar table (for numeric variables such as A or B3), the scalar array table (for numeric arrays -- A(1), B(3,4)), the string symbol table (A$, B2$), and the string array table (B$(2)). Location CDFIO of field 0 contains a CDF to the symbol table field.

2.2.1.1 **The Scalar Table** - The scalar table, the highest table in memory, contains an entry for each numeric variable used in the program. Each entry consists of a three-word floating-point number. The table reserves a few extra entries for temporary results. Location SCSTRT in field 0 contains a pointer to the start of the scalar table.

The Scalar Table



2.2.1.2 **The Array Symbol Table** - The array symbol table consists of successive four-word entries. Each entry specifies the location and size of a numeric array used in the program and has the following format:

Word 1	POINTER TO A(0,0)
Word 2	CDF TO FIELD OF A(0,0)
Word 3	DIMENSION 1
Word 4	DIMENSION 2

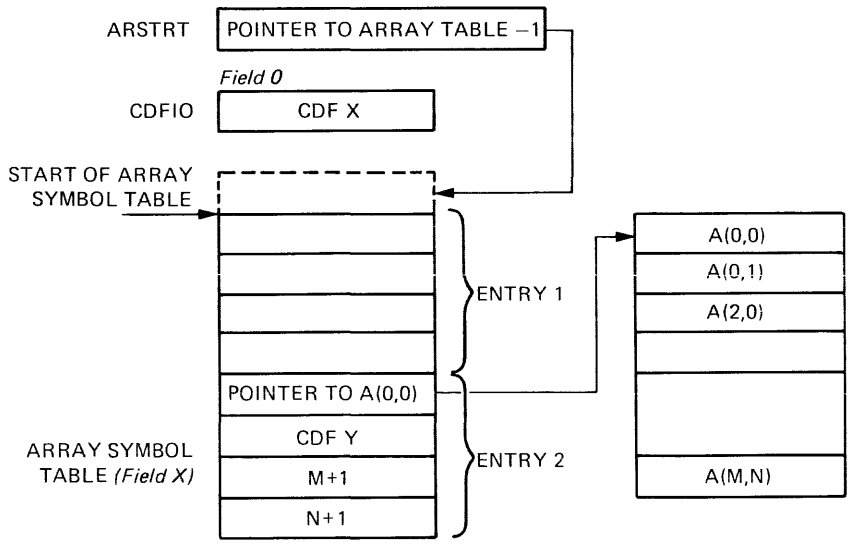
- Word 1 of each entry is a 12-bit pointer to the location of the exponent word of the first element in the array.
- Word 2 is a CDF n where n is the field for the pointer in the first word.
- Word 3 is the first dimension of the array -- obtained by adding 1 to the M in a DIM A(M,N) statement, since the first subscript is always 0.
- Word 4 is the second dimension of the array. If the array is one-dimensional, the second dimension is 0.

To locate the nth element in an array, BRTS performs the following calculation:

Addr of A(M,N)=3*[M+(DIM1 + 1)*N] + Addr of A(0,0)

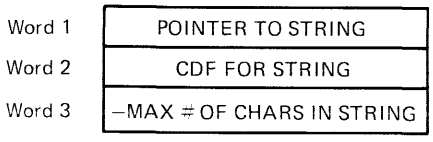
A pointer to the start of the array symbol table less one (for use in an index register) resides in field 0 at location ARSTRT.

The Array Symbol Table



BRTS stores numeric arrays in memory as successive three-word entries with the first subscript varying the fastest and A(0,0) occupying the lowest address in memory.

2.2.1.3 **The String Symbol Table** - The string symbol table contains successive three-word entries in the following format:

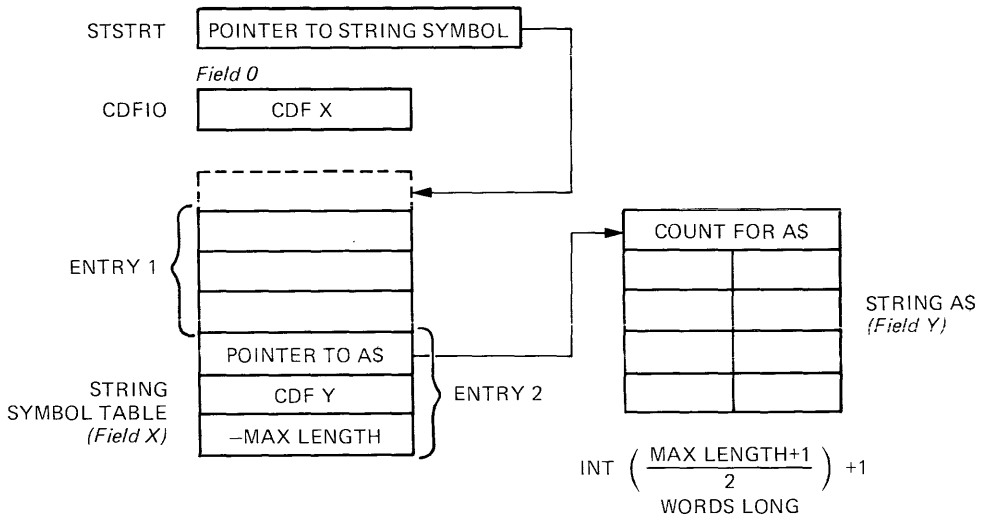


- Word 1 is a 12-bit pointer to the count word of the string.
- Word 2 in the entry is a CDF for the count word.
- Word 3 is the maximum length of the string (in characters) stored as a two's complement negative number. (Each string is allocated INT((n + 1)2)+1 words, where n is the maximum length specified in a DIM statement, whether that many words are actually used or not.)

Note that the maximum number of characters in the string represents the amount of space allocated for the string. The amount of space actually used is represented by the count word, which BRTS stores with the string.

Location SRSTRT in field 0 contains a pointer to the start of the string symbol table (less one).

The String Symbol Table



2.2.1.4 **The String Array Table** - The string array table consists of consecutive four-word entries in the following format.

Word 1	POINTER TO AS(0)
Word 2	CDF FOR AS(0)
Word 3	–MAX = OF CHARS IN AS(0)
Word 4	DIMENSION OF AS(0)

- Word 1 contains a pointer to the count word of string A$(0).
- Word 2 contains a CDF for the count word pointer.
- Word 3 is a two's complement negative number that specifies the maximum length (in characters) of each element in the array.
- Word 4 indicates the size of the string array, obtained by adding 1 to M in the statement DIM A$(M,N) since the first element is always A$(0). A pointer to the start of the string array table less one resides in field 0 at location SASTRT.

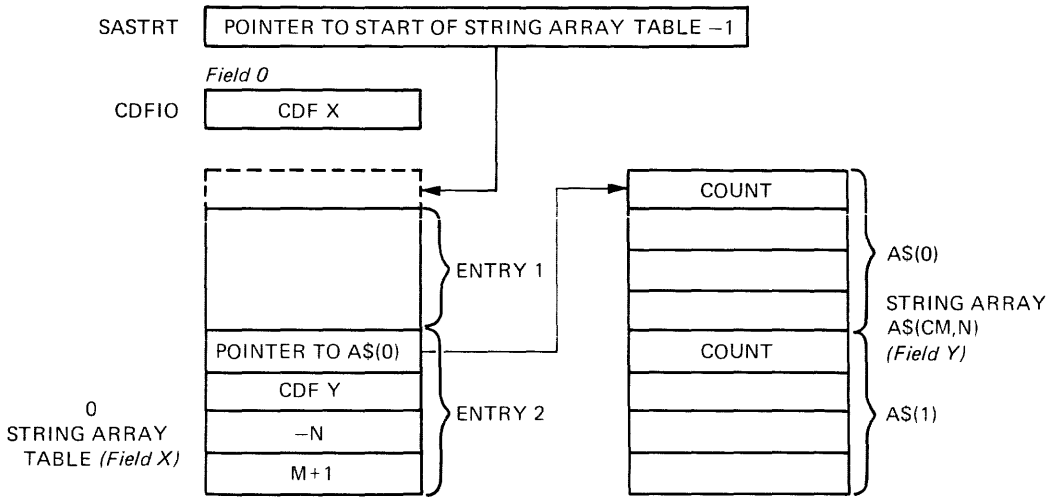
To locate the nth element of a string array, BRTS performs the following calculation:

addr of A$ (N)=addr of A$ (0)+(INT(ABS(Z)+1)/N+1)*N

where

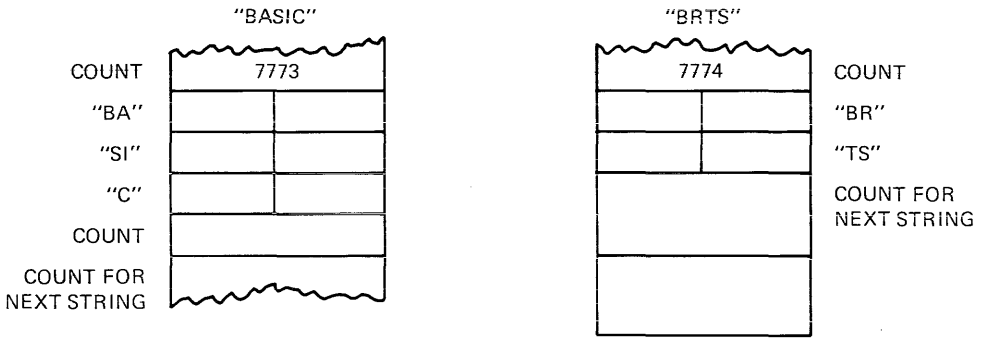
Z = individual character length.

The String Array Table



2.2.2 String Storage

BRTS stores strings as 6-bit ASCII characters. The first word in each string is a character count -- a signed, two's complement number representing the actual number of characters in the string, not the number of words devoted to the string. BRTS fills the left half of each word first, padding out the unused characters with spaces. The minimum string is one character long.

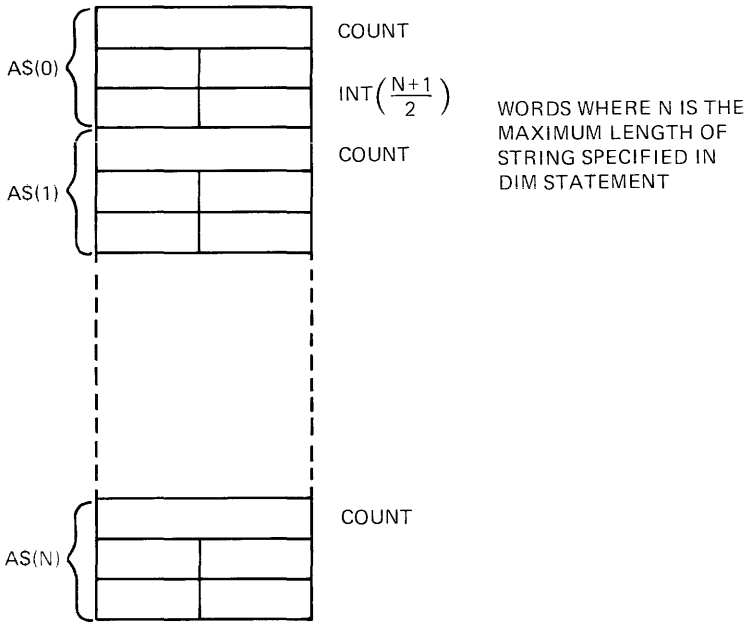


2.2.3 The String Accumulator

BRTS maintains a string accumulator (SAC) for all string operations. String operations leave their results in the SAC and use it as one of their operands. The SAC starts at location SAC in BRTS; it is 80 words long and contains one 6-bit character per word. BRTS stores the length as a negative number in SACLEN and maintains a page 0 pointer (less one) to the start of the SAC at SACPTR.

2.2.4 String Array Storage

BRTS stores string arrays in memory as successive strings, with A$(0) occupying the lowest core address. BRTS allocates space for the maximum length possible, even though not all of the space may be used. The space is for the maximum length.



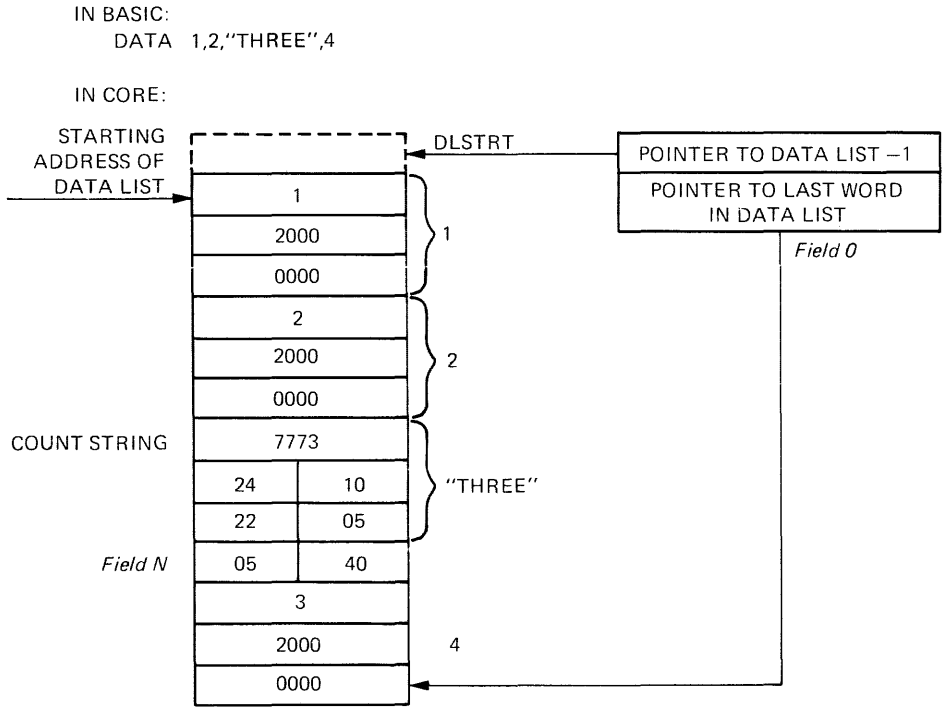
NOTE

For any of the above data types, a field boundary may fall anywhere within any individual item. If your routines use successive words in any data item they must check for a field boundary within that item.

2.2.5 The DATA List

BRTS stores the DATA list (created by the BASIC DATA statement) as sequential items in the highest field in memory. BRTS allocates strings an even number of words and assigns a count word as a prefix.

The DATA list always resides in the highest memory field. BRTS maintains a page 0, field 0 pointer to the starting address of the DATA list less one at DLSTRT. Location DLSTP contains the address of the last word in the list.



2.2.6 Array Space

BRTS reserves space for arrays in the highest memory field. The bottom of the array space (line A in Figure 2-1) can exceed the field boundary and proceed into lower fields, but this happens only in large programs.

2.2.7 Compiler Pseudo-Code

BRTS sends the pseudo-code generated by the BASIC compiler to the highest field in memory. Note that if the bottom of the pseudo-code extends below line B (12000) in Figure 2-1, the file space diminishes, causing a loss in run-time file capabilities. As the bottom of the pseudo-code approaches 10000, the number of files that you may simultaneously open at run time approaches zero. (Each file opened at run time requires at least 400 words of buffer space.)

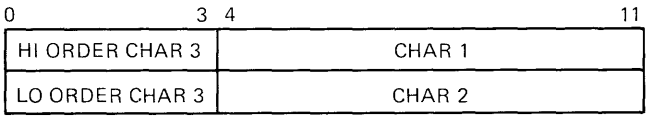
2.2.8 File Buffer Space

BRTS reserves locations 10000-12000 for file buffer space. It allocates buffers as it needs them, starting with the lowest free buffer. BRTS maintains a map of currently allocated buffers called BMAP on page 0. Bits in the map are set if the buffer is in use, and cleared if the buffer is free. Bit 11 represents the buffer from 10000 to 10377, bit 10 for 10400 to 10777, bit 9 for 11000 to 11377, and bit 8 for 11400 to 11777. If any of the buffers are not available because the pseudo-code or variable space extends below 12000, BRTS sets the corresponding BMAP bits at run time.

BASIC files have the following format:

- Numeric files -- store data as consecutive 3-word floating-point numbers, 85 to each 256-word block. The last word in each block is unused. There is no end-of-file marker.

- ASCII files -- store data in OS/8 ASCII format. Three 8-bit characters are packed to every two words in the following manner:



  The end-of-file is marked with a CTRL/Z character.

2.2.9 Device Handler Space

BRTS reserves locations 7000-7577 for 1-page and 2-page device handlers and maintains a map of the 3 pages at DMAP. Bit 11 represents page 7000-7177, bit 10 represents page 7200-7377, and bit 9 page 7400-7577.

Assembly-language functions in programs that do not require more than one or two files open at a time may use some of this handler and file buffer space for their own purposes. You can allocate this space by setting appropriate bits in BMAP and DMAP. After you set the bits, BRTS will not use the space indicated in subsequent FILE operations.

2.2.10 The BRTS I/O Table

BRTS maintains an I/O file table to keep track of the status of each of the four files that may be open simultaneously in a BASIC program. The table contains four 13-word entries, labeled FILE1, FILE2, FILE3, and FILE4, in that order. Each name corresponds to the number you specify in the file statement that opened the file, and each entry has the following format:

```
HEADER WORD
STARTING ADDRESS OF BUFFER (IN FIELD 1)
CURRENT BLOCK IN BUFFER
READ/WRITE POINTER INTO BUFFER
HANDLER ENTRY POINT
STARTING BLOCK NUMBER FOR FILE
ACTUAL FILE LENGTH
MAXIMUM FILE LENGTH
POSITION OF PRINT HEAD (FOR COLUMN FORMAT-
TING)
FILE NAME
FILE NAME
FILE NAME
FILE NAME
```

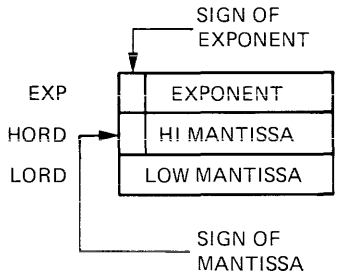
The header word bits have significance as follows:

Bit Positions	Meaning
0-3	OS/8 number for device
4-5	Current character number for unpacking ASCII files
6	0 if the current buffer load has not been changed 1 if current buffer load has been altered
7	0 if device is file structured 1 if device is read/write only
8	0 if the handler is 1 page long 1 if it is a 2 page handler
9	0 if file is fixed length 1 if variable length
10	0 if more data in file 1 if EOF has been seen
11	0 if file numeric 1 if file ASCII

2.2.11 The BRTS Floating-Point Package

The floating-point package is permanently resident in memory and available for use by assembly language routines for floating-point calculations.

2.2.11.1 **The Floating-Point Accumulator** - The floating-point accumulator, FAC, resides at locations EXP, HORD, and LORD on page 0 and has the standard PDP-8 23-bit floating-point format.



Floating-point operations use the FAC in the same way that PDP-8 machine-language instructions use the hardware accumulator. The FAC is one of the operands in every floating-point calculation and holds the result of all floating-point operations (with the exception of FPUT -- see below).

2.2.11.2 **Floating-Point Routines** - BRTS provides the following floating-point routines which you may use as subroutines in a program (For information on calling these routines, see Section 2.3.):

Function	Starting Address	Operation
ADD	FFADD	FAC<-FAC+OPERAND
SUBTRACT	FFSUB	FAC<-FAC-OPERAND
MULTIPLY	FFMPY	FAC<-FAC*OPERAND
DIVIDE	FFDIV	FAC<-FAC/OPERAND
INVERSE SUBTRACT	FFSUB1	FAC<-OPERAND-FAC
INVERSE DIVIDE	FFDIV1	FAC<-OPERAND/FAC
LOAD FAC	FFGET	FAC<-OPERAND
STORE FAC	FFPUT	OPERAND<-FAC

There are also four simple floating-point operations that operate on the FAC and are available to user subroutines.

Function	Starting Address	Operation
NEGATE	FFNEG	FAC<-FAC
NORMALIZE	FFNOR	NORMALIZE<-FAC
SQUARE	FFSQ	FAC<-FAC*FAC
CLEAR	FACCLR	FAC<-0

The functions are called with a JMS and return with the hardware AC=0.

2.2.12 BRTS Overlay Buffer

BRTS allots locations 3400-4577 of field 0 as an overlay area, reading in overlays as it needs them. The overlays, which consist mainly of functions infrequently used, are organized in the following manner:

BASIC.AF Arithmetic Functions

SIN, COS, ATN, EXP, FIX, FLOAT, INT, RND, EXPONENTIATION, SGN, SQR, LOG

BASIC.SF String Functions

ASC, CHR$, DAT$, LEN, POS, SEG$, STR$, VAL, Error processing, TRC

BASIC.FF File Functions

CHAIN, CLOSE, FILE, STOP

BASIC.UF User Function

BRTS reserves the last overlay, BASIC.UF, for user-written assembly language routines. Each time you call for one of these routines, BRTS reads BASIC.UF into the overlay buffer.

2.3 CALLING FLOATING-POINT ROUTINES

There are two separate calling sequences for floating-point routines:

- Mode 1, which you use when the operand of the routine is in field 0 (the same field as the FPP).
- Mode 2, which you use when the operand is in some other field.

The contents of the hardware accumulator at the time of entry also determine the mode of the calling sequence. You may use Mode 1 only if the accumulator is 0. If the AC is non-zero, you must use Mode 2.

You set a switch in the calling sequence -- location FF -- to tell the floating-point package which mode to follow. For Mode 1, let FF equal zero; for Mode 2, non-zero.

In a Mode 1 call, the address of the operand immediately follows the JMS instruction. Thus:

```
        CLA
        DCA FF          /SWITCH FF=0 FOR MODE 1
        JMS I POINTR    /JUMP TO FLOATING-POINT ROUTINE
        (OPERAND ADDR)  /12-BIT ADDRESS OF OPERAND
        .               /RETURNS HERE
        .
        .
POINTR, (STARTING ADDR)
                        /FLOATING-POINT ROUTINE
                        /STARTING ADDRESS.
```

In a Mode 2 call, the address of the operand is in the accumulator. The CDF n instruction indicates the field of the operand. For example,

```
        CLA IAC
        DCA FF              /FF SWITCH NOT EQUAL TO 0 FOR MODE 2
        CDF N               /DF TO FIELD OF OPERAND
        TAD OPADDR          /ADDRESS OF OPERAND
        JMS I POINTR        /JUMP TO FLOATING-POINT ROUTINE
        (UNUSED)            /THIS LOCATION UNUSED
        .                   /RETURNS HERE.
POINTR, (STARTING ADDR)     /ADDRESS OF FLOATING-POINT ROUTINE
OPADDR, (OPERAND)           /ADDRESS OF OPERAND
```

Both modes return with a clear AC and the data field set to 0. Note that the routine does not alter switch FF. Therefore, it is necessary to change it only when you want to change modes, not before every call.

Both modes return to the second instruction following the JMS call, skipping the word immediately after the JMS. Since a Mode 2 call never uses this location, you may use it as a location for storing constants in Mode 2 operations.

The FF switch -- which might seem unnecessary in most calling sequences -- makes it possible for the floating-point package to obtain an operand for location 0 in a field other than zero. If you did not include the FF switch, the FFP would examine the accumulator, find it empty, and use the address in the word following the call, since it has no way to tell an empty AC from an AC containing an operand address of 0. The FF switch, then, simply tells the floating-point package whether the zero means "Mode-1 call" or "operand at 0."

BRTS contains Page 0 literals used by the FGET and FPUT routines. These Page 0 literals can be found in the BRTS source listing. Page 0 literals reference the following routines (For more information on Page 0 literals, refer to the section on BRTS subroutines.):

Page Zero Link	Routine
FNEGL	FFNEG
FNORL	FFNOR
FCLR	FACCLR

The following sample programs demonstrate uses of floating-point routines.

1. This routine calculates X^2+2X+1.

```
            .
            .
            CLA
            DCA FF              /OPERAND ADDRESS WILL
                                /FOLLOW CALLS (MODE 1)
            JMS I FGETL         /LINK IS ON PAGE 0
            X
            JMS I FMPYLK        /X * X
            X
            JMS I FPUTL         /SAVE X^2
            Y
            JMS I FGETL         /LOAD X AGAIN
            X
            JMS I FMPYLK        /2X
            TWO
            JMS I FADDLK        /2X+1
            ONE
            JMS I FADDLK        /X^2+2X+1
            Y
            .
            .                   /RESULT NOW IN FAC
            .
FADDLK,     FFADD               /LINK TO ADD ROUTINE
FMPYLK,     FFMPY               /LINK TO FLOATING MULTIPLY
TWO,        0002                /FLOATING-POINT CONSTANT
            2000                /2.0
            0000
ONE,        0001                /FLOATING-POINT CONSTANT
            2000                /1.0
            0000
X,          ...                 /VARIABLE
            ...
            ...
Y,          0                   /FLOATING-POINT TEMPORARY
            0
            0
```

2. This routine adds five successive floating-point numbers starting at location 0 field 2.

```
START,      CLA
            DCA OPADDR          /FIRST OPERAND AT LOCATION 0
            JMS I FCLR          /ZERO FAC
            IAC
            DCA FF              /CALLS ARE MODE 2
ALOOP,      CDF 20
            TAD OPADDR          /OPERAND ADDR IN AC
            JMS I FADDLK        /CALL ADD ROUTINE
MINUS5,     -5                  /LOCATION UNUSED, SO WE USE
            TAD OPADDR          /IT AS A COUNTER

            TAD K3              /UPDATE OPERAND ADDRESS
            DCA OPADDR
            ISZ MINUS5          /DONE?
            JMP ALOOP           /NO
            HLT                 /YES-ANSWER IN FAC.
FADDLK,     FFADD               /POINTER TO ADD ROUTINE
OPADDR,     0                   /POINTER TO OPERAND
K3,         3                   /EACH OPERAND IS 3 WORDS LONG.
```

2.4 USING BRTS SUBROUTINES IN ASSEMBLY-LANGUAGE FUNCTIONS

BRTS includes several subroutines that you may use in assembly language functions. In the following discussion, each subroutine has a symbolic tag for its starting address. These tags can be found in the symbol table. Many routines are now addressed with Page 0 literals that can be found in the BRTS source listing. Note that references to Page 0 pointers by name no longer apply. The purpose is to shorten the size of the BRTS symbol table.

2.4.1 ARGPRE

ARGPRE locates scalar variables in the scalar table. You can use it to pass arguments to and from a user subroutine. When you call it, ARGPRE reads the rightmost eight bits (0-255 decimal) of location INSAV as the position of the item you wish to locate in the array. For example, if you place a 2 in INSAV, ARGPRE will locate the third variable in the scalar table. (The first entry is zero.) On return, ARGPRE sets the data field to the field of the variable and leaves the location of the exponent word of the variable in the accumulator. To call ARGPRE, use a JMS instruction.

For example, the following assembly-language sequence -- which includes a call to the ARGPRE subroutine -- loads the third variable in the scalar table into the floating-point accumulator.

```
            CLA
            TAD C2              /WE WANT ENTRY #3, BUT
                                /SINCE THE FIRST ONE IS 0,
                                /LOAD INSAVE WITH 2
            DCA INSAVE
            IAC
            DCA FF              /SET FF SWITCH
            JMS I ARGPRL        /CALL ARGPRE
            JMS I FGETL         /THE AC AND DATA FIELD
            (UNUSED)            /ARE SET, SO THIS IS A
            HLT                 /MODE 2 CALL.
C2,         2
ARGPRL,     ARGPRE
```

2.4.2 XPUTCH

XPUTCH reads an ASCII character from the accumulator and loads it into the terminal ring buffer. To use XPUTCH, place an ASCII character in the rightmost eight bits of the accumulator and call for the subroutine with a JMS instruction.

On return, XPUTCH clears the accumulator. Note that XPUTCH does not print the character; it simply puts the character in the ring buffer.

For example, this sequence uses XPUTCH to place a carriage return/line feed combination into the terminal buffer:

```
            CLA                 /LOAD CR INTO AC
            TAD K215            /CALL XPUTCH VIA PAGE 0 LINK
            JMS I XPUT          /LOAD LINE FEED INTO AC
            TAD K212            /PUT IN BUFFER
            JMS I XPUT
            HLT
K215,       215                 /ASCII CODE FOR CR
K212,       212                 /ASCII CODE FOR LF
```

2.4.3 XPRINT

Subroutine XPRINT prints the next character in the ring buffer. If the ring buffer contains characters waiting to be printed, XPRINT returns to the instruction following the JMS that called it. If the buffer is empty, XPRINT skips the instruction immediately following the JMS. XPRINT will print a character only if the terminal is not busy, so that a call to XPRINT means "print a character if possible" rather than "print a terminal character."

The call to XPRINT in the following example keeps the terminal busy during a compute-bound loop. At the end of the loop, XPRINT empties the ring buffer.

```
LOOP,   .
        .               /COMPUTING INSTRUCTIONS
        .
        JMS I PRINT     /CALL XPRINT VIA PAGE 0 LINK
        NOP             /THIS INSTRUCTION WILL BE
                        /SKIPPED IF RING BUFFER IS EMPTY
        .
        .
        ISZ LOOPCN      /LOOP CONTROLLING INSTRUCTION
        JMP LOOP        /
        JMS I PRINT     /LOOP IS DONE - EMPTY RING
        JMP .-1         /BUFFER BEFORE CONTINUING
```

2.4.4 PSWAP

Under normal conditions, BRTS runs with the OS/8 page 17600 portion of the resident monitor moved to the highest page of memory (the second-highest page in a TD8/E system). PSWAP lets you restore this page to 17600 prior to doing any operations with OS/8 and then swap it back up to high memory when you are through. Note that this means you must always use PSWAP an even number of times.

The following sequence of code -- which directs the USR in OS/8 to perform a lookup on file BASIC.DA -- requires two JMS calls to PSWAP.

```
        .
        .
        .
        CLA             /AC SHOULD BE 0 ON CALL
        JMS I P1SWAP    /RESTORE OS/8 PAGE 17600 RESIDENT
        CLA IAC         /DEVICE # FOR SYS: IS 1
        CIF 10
        JMS I K7700     /CALL USR
        2               /LOOKUP
        FNAME           /POINTER TO FILE NAME
        0               /CONTAINS LENGTH ON RETURN
        HLT             /ERROR RETURN
        JMS I P1SWAP    /SWAP OS/8 RESIDENT BACK
        .               /TO HIGH CORE
        .
        .
```

2.4.5 UNSFIX

UNSFIX fixes a positive, 12-bit, magnitude-only integer from the floating-point accumulator and returns with the result in the hardware accumulator. UNSFIX destroys the contents of the FAC.

The range of the fixed integer is 0-4095. Any attempt to fix a number larger than 4095 or a negative number will produce an "FO" or "FM" error message, respectively. To call UNSFIX, use a JMS instruction.

The following code -- which includes a call to INFIX via INTL -- uses the FAC as a counter for the number of times to ring a bell on the terminal.

```
        .
        .
        .
        CLA
        JMS I INTL      /FIX THE FAC TO 12-BIT INTEGER
        CIA             /NEGATE THE INTEGER
        DCA COUNTR      /AND STORE AS COUNT
BELLOP, TAD K207        /ASCII FOR BELL
        JMS I XPUT      /PUT IN RING BUFFER
        ISZ COUNTR      /RIGHT NUMBER YET?
        JMP BELLOP      /NO-RING ANOTHER BELL
        .
        .
        .
K207,   207
```

2.4.6 STFIND

Depending on the contents of the link bit, STFIND locates a string variable or the first element in a string array.

- If you set the link to 0, STFIND accepts the rightmost eight bits of location INSAV as the position of the variable you wish to locate in the string symbol table.

- If you set the link at non-zero, STFIND accepts the rightmost five bits in INSAV as a position in the string array table.

After STFIND returns, the AC contains a CDF to the field of the string specified; location STRPTR points to the first word -- the count word -- of the string; location STRMAX holds the maximum length of the string as a negative number; and STRCNT contains the actual number of characters in the string as a negative number. STFIND is used most frequently to pass arguments to and from user functions.

The following sequence uses STFIND to locate string number seven:

```
        TAD K6          /THE NUMBERING STARTS WITH 0
        DCA INSAV       /SET UP STFIND POINTER
        CLL             /WE WANT SIMPLE STRING
        JMS I STFINL    /CALL STFIND
        .
        .
K6,     6
STFINL, STFIND
```

This example locates the first element of string array number two:

```
        TAD K1          /THE SECONDENTRY
        DCA INSAV
        CLL CML         /WE WANT STRING ARRAY
        JMS I STFINL    /CALL STFIND
        .
        .
K1,     1
STFINL, STFIND
```

2.4.7 MPY

MPY performs a 12-by-12-bit multiplication. It multiplies the contents of the hardware accumulator by the contents of location TEMP3 (both numbers are treated as 12-bit unsigned integers). On return, MPY stores the high-order bits of the result in TEMP6 and the low-order bits in the AC.

2.4.8 DLREAD

DLREAD places the next word in the data list into the accumulator. If the list contains no more data, a DA error message results.

The following sequence of instructions reads a number from a DATA list into the hardware accumulator:

```
        CLA
        JMS I DLREAL    /READ EXPONENT WORD INTO AC
        DCA EXP         /STORE IN FAC
        JMS I DLREAL    /READ HIGH MANTISSA FROM LIST
        DCA WORD        /STORE HIGH MANTISSA WORD
        JMS I DLREAL    /READ LOW MANTISSA FROM LIST
        DCA LORD        /STORE LOW MANTISSA WORD
        .
        .
        .
DLREAL, DLREAD
```

2.4.9 ABSVAL

ABSVAL determines the absolute value of the floating-point accumulator. If the FAC is negative, ABSVAL negates it before return. If the FAC is positive, ABSVAL is the equivalent of a NOP.

2.5 PASSING ARGUMENTS TO THE USER FUNCTION

You call for a user assembly-language function with a JMS. Before BRTS executes the instruction, it places the first numeric argument of the function in the floating-point accumulator, the second in entry 0 of the scalar table, the third in entry 1, and so on through the list of arguments. If the function uses string arguments, BRTS places the first in the string accumulator, the second in entry zero of the string table, the third in entry 1, and so on. The function obtains these arguments as it needs them by calling ARGPRE and STFIND.

For example, the following function takes the first two numeric arguments and performs on them the operation specified in the string argument, A$:

```
UDEF EXM (X,A$, Y)
LET Z=EXM (2,"PLUS",1)
```

Legal values for A$ are strings beginning with "PL" for "PLUS" and "MI" for "MINUS". Thus:

```
EXM,     0              /ENTRY POINT
                        /INDEX REGISTER 5 POINTS TO SAC
         TAD I SACP     /GET FIRST 2 CHARS OF A$ FROM SAC
         TAD PL         /COMPAR THEM TO "PL"
         SZA CLA
         JMP EMINUS     /NOT "PLUS"-CHECK FOR "MINUS"
         DCA INSAV      /OPERATION IS PLUS-INIT ARGRE TO GET
                        /SCALAR 0
         JMS I ARGPRL   /FIND Y. X IS ALREADY IN FAC
         JMS I FADDL    /X+Y
ARGPRL,  ARGPRE         /THIS LOC SKIPPED BY FADD
         JMP I EXM      /DONE-RETURN WITH RESULT IN FAC
EMINUS,  TAD I SACP     /FIRST TWO CHARS OF SAC AGAIN
         TAD MI         /COMPAR TO MI
         SZA CLA        /IS IT "MINUS"?
         JMP I IAL      /NO-ERROR
         DCA INSAV      /YES-SET UP ARGPRE FOR ENTRY 0
         JMS I ARGPRL   /FIND Y. X IS ALREADY IN FAC
         JMS I FSUBL    /X-Y
SACP,    SAC            /THIS LOC SKIPPED BY FSUB
         JMP I EXM      /RETURN WITH VALUE IN FAC
PL,      -2014
MI,      -1511
FADDL,   FFADD
FSUBL,   FFSUB
IAL,     IA
```

If the function returns a value, it should leave it in the floating-point accumulator. The function returns with a JMP I through the entry point. (If you enter a JMP to location IA in BRTS, this will generate a fatal IA -- an illegal argument.)

2.5.1 Using the USE Statement

If the assembly-language function needs to know the location of an array (for buffer space, multiple argument passing, array argument), you must use the USE statement. The USE statement places the octal number for the array specified into location USECON. By using this value as an index into the array symbol table, the function can locate the data it requires.

For example, the hypothetical assembly-language function PLT requires a 100-word buffer. To assure allocation of this buffer when you use the PLT function in a BASIC program, you must create a 34-element array and identify it with a USE statement before calling the PLT function. Thus:

```
10 REM DEFINE THE USER FUNCTION
20 UDEF PLT (X,Y)
30 REM ALLOCATE A 34 ELEMENT (102 WORDS) ARRAY FOR A BUFFER
40 DIM B(34)
      .
      .
      .
```

The function PLT finds B as follows:

```
      .
      .
      .
100 USE B
110 Y=PLT(3,2.8)
      .
      .
      .
```

```
PLT,    0
        TAD USECOND     /GET ENTRY NUMBER OF B
        CLL TRL         /MULTIPLY BY 4 (EACH ARRAY TABLE ENTRY
                        /IS 4 WORDS LONG)
        TAD ARSTRT      /MAKE POINTER INTO ARRAY TABLE
        DCA XR5         /AND SAVE IT
        TAD CDFIO       /GET CDF TO SYMBOL TABLE FIELD
        DCA /+1         /PUT INTO LINE
        .               /CHANGE DF TO SYMBOL TABLE FIELD
        TAD I XR5       /GET POINTER TO B(0)
        DCA BPTR        /SAVE FOR LATER
        TAD I XR5       /GET ARRAY DIMENSION 1
        DCA DIM1
        TAD I XR5       /GET ARRAY DIMENSION 2
        .
        .
        .
```

Note that the USE statement simply passes an array entry number to the assembly-language function. The function must obtain all actual parameters from the array symbol table, using that entry number as an index. Note also that the arrays passed in such a fashion may reside almost anywhere in memory and that a field boundary may fall within the array.

2.6 BRTS INPUT/OUTPUT

BRTS drives the terminal asynchronously by maintaining a 40-character terminal output ring buffer and regularly calling subroutine XPRINT. It operates in the following manner:

- BRTS calls subroutine XPUTCH, which inserts characters into the terminal ring buffer. If the ring buffer is full, XPUTCH waits until BRTS calls XPRINT to print a character, opening up a place.
- BRTS regularly calls XPRINT (at least once every pseudo-instruction). XPRINT works in the following manner:
    - If the terminal flag is not set, XPRINT returns.
    - If the flag is set, XPRINT checks the buffer for more characters. If it finds a character, it prints it (with a TLS) and returns.

If the ring buffer contains characters waiting to be printed, XPRINT returns to the instruction immediately following the JMS that called it. If the ring buffer is empty, XPRINT skips the instruction after the JMS upon returning. This technique allows BRTS to do other things for most of the one hundred milliseconds without turning on the interrupt facility. Although this method requires periodic calls to XPRINT, it still consumes considerably less time than waiting for the terminal flag.

Assembly language functions may use the ring buffer (BRTS empties it before it calls the function), or they may perform simple terminal I/O with TLS, TSF, and JMP.-1 instructions. If a function does not use the ring buffer, it must make sure that the terminal flag is set before it returns to BRTS.

Note that an assembly language function does not have to call XPRINT. It may place a character in the ring buffer and let XPRINT take care of it on its next regular call from BRTS.

2.7 INTERFACING AN ASSEMBLY LANGUAGE FUNCTION TO BRTS

You call an assembly language function the same way you call any other subroutine -- with a JMS instruction. The JMS causes BRTS to use the symbolic address of the function to look up its actual location in the user function table. This table, which begins at 1560 in BRTS, contains absolute pointers to the starting address of each user assembly language function. You must place all user functions between 3400 and 4577, the space which BRTS reserves for the user function overlay, BASIC.UF. User functions must return to BRTS via a JMP I through their starting address.

To run a set of user assembly language functions under BRTS, you must perform the following operations:

1. Assemble all the user assembly language functions together. You may include up to sixteen functions. They must fit between 3400 and 4577 but may reside anywhere within that space.

```
.R PAL8
*USER.BN<USER.PA
```

2. Load the user functions into memory with the absolute loader (ABSLDR) and SAVE locations 3400-4577 as the file BASIC.UF, which is the user overlay.

```
.R ABSLDR
*USER.BN$
.SAVE SYS:BASIC.UF 3400-4577
```

3. Modify the user function table in BRTS with ODT, entering absolute pointers for the starting addresses of the functions. All unmodified locations in the table contain a value of 240 octal. Replace this value with the starting address pointer. Start at location 1560 and enter the pointers in the same order in which the functions appear in the UDEF statement that defines them.

```
.GET SYS:BRTS
.ODT
1560/240 3400
1561/240 3410
^C
.SAVE SYS:BRTS
```

This procedure interfaces two functions that start at locations 3400 and 3410 respectively.

For example, the following package contains three assembly language functions: HI, PLT, and LO. You may define these in any order in the DEF statement as long as you remember to enter them in the same order in the user function table.

```
        *3400
HI      0               /ENTRY POINT FOR HI
        .               /ORDER OF ENTRY POINTS IS
        .               /NOT CRITICAL
        .
        JMP I HI
PLT,    0               /ENTRY POINT FOR PLT
        .
        .
        .
        JMP I PLT
LO,     0               /ENTRY POINT FOR LO
        .
        .
        .
        JMP I LO
```

To enter these three functions into the user function table, follow this procedure:

```
.GET SYS:BRTS
.ODT
1560/240 PPPP
1561/240 HHHH
1562/240 LLLL
^C
.SAVE SYS:BRTS
```

where PPPP, HHHH, and LLLL represent octal starting addresses for PLT, HI, and LO respectively.

BRTS sets up a one-to-one correspondence between the pointers at 1560 and the function names in the UDEF statement for the package. Therefore, the order of the pointers must correspond exactly to the order of the function definitions in UDEF. If you wish to use only the nth function in a given user package, you must still define n functions in the UDEF statement, although the first n-1 may be dummies.

For example, consider a package of eight assembly language functions listed in the user function table in the following order:

```
ONE (X)
TWO (X,Y)
THR (X,Y,Z)
FOU (X,Y,Z,A)
FIV (X,Y,Z,A,A$)
SIX (X,Y,Z,A,A$,B$)
SEV (X)
EIG (Y)
```

If you want to use only function ONE in a BASIC program, your UDEF statement will look like this:

```
10 UDEF ONE(X)
```

If you want to use only functions ONE and EIG, the UDEF might look like this:

```
10 UDEF ONE(X),DUA(D),DUB(D),
DUC(D),DUD(D),DUE(D),DUF(D),
EIG(Y)
```

In this statement, DUA through DUF are dummy user function names that have no effect on the program at run time. They simply set up the right correspondence between names and pointers.

The easiest and surest way to match up all function names and pointers correctly is to write a UDEF statement for every function in the package.

2.8 SOME GENERAL CONSIDERATIONS

2.8.1 Routines Unusable by Assembly Language Functions

Because the assembly language functions reside in the overlay buffer during execution, they cannot use any routines that reside in any of the other three overlays. These routines include:

Routine Name	Function
FFATN	Arctangent Function
FFCOS	Cosine Function
FFEXP	Exponential Function (e^x)
EXPON	Power Function (A^B)
INT	Signed integer Function
FFLOG	Naperian log Function
SGN	Sign Function
FFSIN	Trigonometric Sine Function
RND	Random Number generator
FROOT	Square root Function
ASC	String Function ASC
CHR	CHR$ Function
DATE	DAT$ Function
LEN	String length Function
POS	String search Function
SEG	String segmenting Function
STR	STR$ Function
VAL	VAL Function
TRC	Trace Function
CHAIN	File manipulation Routines
CLOSE	
OPENAF	
OPENAV	
OPENNF	
OPENNV	

2.8.2 Using OS/8

A carefully designed assembly language function -- one that protects all memory areas required by BRTS -- may use OS/8 without restriction. Once PSWAP has swapped the 17600 portion of the resident monitor out of high memory, the assembly-language function may call the User Service Routine and then locate, use, and close files at will.

2.8.3 Using Device Driver and File Buffer Space

If your BASIC program does not need full file capabilities, any assembly-language function in the program may use the driver space from 7000 to 7577 and the buffer space from 10000 to 17777. However, the function must check the bit maps and status words on page 0 before it uses any part of the space to make sure it is available.

2.8.4 Using the Interrupt Facility

OS/8 BASIC runs with the interrupt facility turned off. However, BRTS reserves locations 0-2 on page 0 for any assembly language function

that wishes to use the interrupt. Before turning on the interrupt system, an assembly language function must clear all the flags set by the OS/8 handlers. Before returning, the function must turn off the interrupt and set the TTY flag.

2.8.5 Using Page 0

The following map shows BRTS page 0 usage. An assembly language function may use the locations marked with an asterisk (*) without saving the contents.

Locations	Usage
0-2 *	Interrupt vector
3-7	System parameters and temps
10-15 *	Index registers
16-17	System pointers
20-30	Compiler-BRTS communication
30-36	System registers
37-62	Floating-point package area
63-67	System registers
73-107	Constants
110-161	Links to BRTS subroutines
162-177	I/O Table pointers

Assembly language functions may use any of the pointers or constants on page 0, but they must be intact when control returns to BRTS.

CHAPTER 3

OPTIMIZING SYSTEM PERFORMANCE

You may take advantage of several ways to speed up the time it takes to compile and run an OS/8 BASIC program.

3.1 BYPASSING THE BASIC EDITOR

Running a source program according to standard BASIC procedure is a three-step process. You must:

1. Call the BASIC EDITOR
2. Request the program with an OLD command
3. Run the program with RUN command

For a simpler and speedier method, bypass the BASIC editor and run the program directly with a COMPILE or EXECUTE monitor command. The format is

```
COMPILE indev:file.BA
```

where

.BA is an extension indicating that the input file contains a BASIC source program.

Summoned in this manner, OS/8 BASIC returns control to the Monitor rather than the BASIC editor when it has finished running the program.

As a general rule, use the BASIC editor to:

- create new programs or modify old ones
- debug old programs

and use COMPILE and EXECUTE to:

- run existing programs
- run BASIC programs in BATCH stream

To run with a COMPILE or EXECUTE command, a BASIC source program must conform to the following rules:

- It may contain no blank lines.
- All statements must appear in the order that BASIC will execute them.

If you intend to run your program from the Monitor only, you do not have to begin every line with a line number. Only the lines that you specify as destinations in IF, GOTO, and GOSUB commands require numbering. The following example contains no unnecessary line numbers:

```
    FOR I=1 TO 10
    IF I=2 THEN 400
    PRINT I
    GO TO 410
400 PRINT "TWO"
410 NEXT I
    END
```

Note that the BASIC editor will not accept unnumbered lines. To write and enter a program without numbering every line, you must use the OS/8 Editor or TECO. Experienced users will discover that these editors provide many features not available from the BASIC editor.

3.2 PLACING BASIC OVERLAYS ON THE SYSTEM DEVICE

DECtape users can improve the performance of their system by following these two procedures:

- Use a DECtape drive other than DTA0 for DSK. (See the ASSIGN command.)
- Place the OS/8 BASIC system files as close together on the SYS tape as possible. The best way is to make a "BASIC tape" containing only the OS/8 system, PIP, and the BASIC system image files.

Both procedures speed up OS/8 BASIC by reducing the tape motion required for overlaying and compiling.

3.3 GROUPING FUNCTION CALLS IN BASIC PROGRAMS

Most of the BASIC functions and file operations reside in three system overlays. Since the system overlay driver reads in an overlay only if the function you call for does not reside on the currently resident overlay, you can reduce program execution time simply by grouping calls to functions that reside on the same overlay. For example:

```
10 INPUT A$
20 Z$= SEG$(A$,1,6)
30 FILEN #1: Z$
40 INPUT A$
50 Z$= SEG$(A$,1,6)
60 FILEN #2: Z$
```

This program accepts two strings that you enter at the terminal and reads the first six characters of each as a file name to open a BASIC file. To accomplish this, the program uses the SEG$ function, a file statement, the SEG$ function, and the file statement again. Since SEG$ and the file statement reside on different overlays, the driver must perform four separate operations. The following program produces

the same result more efficiently by grouping function calls and file statements together in such a way that the driver has to operate only twice:

```
10 INPUT A$,B$
20 Z$=SEG$(A$,1,6)
30 X$=SEG$(B$,1,6)
40 FILEN #1: Z$
50 FILEN #2: X$
```

The system overlays distribute the BASIC functions and file operations in the following manner:

Overlay 1 (BASIC.AF):	SIN,COS,ATN,LOG,EXP,RND, SQR,SGN,POWER(A^B)
Overlay 2 (BASIC.SF):	ASC,CHR$,DAT$,LEN,POS, SEG$,STR$,VAL
Overlay 3 (BASIC.FF):	CLOSE,FILE,FILEN,FILEV, FILEVN

3.4 MAKING SAVE IMAGES OF BASIC SOURCE PROGRAMS

Normal BASIC program execution requires a minimum of six file access operations. By contrast, the execution of memory-image files requires no more than two file accesses -- one to read the memory-image file and one to read BRTS if the BCOMP /B option (see below) was not specified. Memory-image file execution also eliminates compiler/loader overhead, thus greatly reducing execution time, especially on DECtape systems.

To create a memory-image file from a BASIC language source program, type

```
.R BCOMP
*DEV:PROG.BA/K
```

where PROG.BA is the source. The K switch indicates that a memory-image file is to be created.

The following BCOMP options apply to SAVE operations.

Option	Meaning
/K	Indicates that a memory-image file will be created.
/N	Indicates that the memory-image program will never be executed on a 12K TD8E system. This saves 400 words of memory but reduces configuration independence.
/B	Loads a copy of the run-time system into the memory image. This increases the size of the memory-image file by 10 to 50 percent (exactly 15 blocks) but eliminates the need for a file access to read in BRTS at run time. BRTS and its overlays must still exist on the system device when the program runs.

=n Indicates the highest field that the program will use (up to 7 octal). Field n must fall in the range 1<n<m, where m is the highest memory field (up to 7) available on the host machine -- that is, the machine on which the program is written. The highest memory field on the target machine -- the machine on which the program will run -- is n. This may reduce configuration independence, since the resulting memory image will not load correctly on a machine with fewer than n+1 memory fields. If n is omitted, n=1. If you specify n larger than m, n=m is assumed.

/C In BCOMP, the /C option is used in conjunction with the /K option to create a file that can be chained to from a non-BASIC file. For example:

```
.R BCOMP
*EXAM.BA/C/K
```

/V In BCOMP, the /V option is used to obtain the current version number of COMP, BLOAD, and BRTS. For example:

```
.R BCOMP
*EXAM.BA/V
```

This causes the system to print at the console the current version numbers for BCOMP, BLOAD, and BRTS as part of the output of the file being compiled.

In the absence of error conditions, the compiler post-processor (BLOAD) will exit to OS/8. At this time, type:

```
.SA DEV:PROG
```

to create an executable memory image. Additional arguments to the SAVE command must not be specified. The memory image is executed by typing:

```
.R PROG
```

or

```
.RUN DEV:PROG
```

The following error messages may occur during execution of a BASIC memory-image file:

USER ERROR 1 AT nnnn

One of the files:

BRTS.SV
BASIC.AF
BASIC.SF
BASIC.FF

was missing from the system device.

USER ERROR 2 AT nnnn

An attempt was made to load a memory-image file produced under the /N option on a 12K TD8E system (without ROM).

USER ERROR 3 AT nnnn

Insufficient memory to load this core image file.

When executing BASIC memory-image files on a DECtape system, the following techniques will ensure minimum execution time:

- Follow the recommended procedure for grouping calls to functions according to the overlay in which the function resides, to minimize overlaying at run time.

- Prepare a system DECtape that contains all of the BASIC memory-image files, followed by:

```
BRTS.SV
BASIC.AF
BASIC.FF
BASIC.SF
BASIC.UF (optional)
```

The BASIC memory-image files should reside near the beginning of the DECtape. If chaining is employed, the least frequently run programs should appear first on the DECtape.

CHAPTER 4

OS/8 BASIC SYSTEM BUILD INSTRUCTIONS

4.1 THE BASIC SYSTEM

OS/8 BASIC is distributed on DECtape and paper tape. The DECtape version contains SAVE images for each of the OS/8 BASIC system components as well as binaries. The paper tape distribution includes binaries for each of the system components. OS/8 BASIC comprises the following files.

File	Component
BASIC.SV	Editor save image
BCOMP.SV	Compiler save image
BLOAD.SV	Loader save image
EABRTS.SV	KE8/EAE version of Run-time System save image
EAEOVR.BN	Overlay for KE8/E EAE (8/E with KE-8E-EAE)
BRTS.SV	Run-time System save image
BASIC.AF	Arithmetic function overlay
BASIC.SF	String function overlay
BASIC.FF	File manipulation overlay

4.2 MAKING SAVE IMAGES FROM BINARY FILES

4.2.1 Non-EAE BASIC

To create SAVE images for each of the OS/8 BASIC binary files, use the following build procedure for OS/8 BASIC (non-EAE). All system programs must reside on the system device -- SYS:.

1. For the Editor:

```
.PAL BASIC
.LOAD BASIC
.SAVE SYS:BASIC;3211
```

2. For the Compiler:

```
.PAL BCOMP
.LOAD BCOMP
.SAVE SYS:BCOMP;7000
```

3. For the Loader:

```
.PAL BLOAD
.LOAD BLOAD
.SAVE SYS:BLOAD;7605
```

4. For the Run-time System:

```
.PAL BRTS/W
.LOAD BRTS
.SAVE SYS:BRTS 0-6777;7605
.SAVE SYS:BASIC.AF 3400-4577;7605
.SAVE SYS:BASIC.SF 12000-13177;7605
.SAVE SYS:BASIC.FF 13400-14577;7605
```

5. At this point, BASIC is ready to run.

4.2.2 EAE BASIC

Use the following procedure to create SAVE image files for OS/8 BASIC EAE. Note that all system programs must reside on the system device -- SYS:.

1. For the Editor:

```
.R PAL8
*DEV:BASIC.BN<DEV:BASIC.PA
.R ABSLDR
*DEV:BASIC.BN$
.SAVE SYS:BASIC;3211
```

2. For the Compiler:

```
.R PAL8
*DEV:BCOMP.BN<DEV:BCOMP.PA
.R ABSLDR
*DEV:BCOMP.BN$
.SAVE SYS:BCOMP;7000
```

3. For the Loader:

```
.R PAL8
*DEV:BLOAD.BN<DEV:BLOAD.PA
.R ABSLDR
*DEV:BLOAD.BN$
.SAVE SYS:BLOAD;7605
```

4. For the Run-time system:

```
.R PAL8
*DEV:EARBRTS.BN<TTY:,SYS:BRTS.PA/W
   (pause)
EAE=1
^Z
   (pause)
^Z
.R ABSLDR
*DEV:EABRTS.BN$
.SAVE SYS:BRTS 0-6777;7605
.SAVE SYS:BASIC.AF 3400-4577;7605
.SAVE SYS:BASIC.SF 12000-13177;7605
.SAVE SYS:BASIC.FF 13400-14577;7605
```

NOTE

All BASIC system files must reside on the system device (SYS).

5. At this point, BASIC is ready to run.

4.3 ASSEMBLING THE BASIC SOURCES

The following instructions show how to assemble each of the BASIC sources with the PAL8 assembler. The descriptions represent OS/8 keyboard commands. To assemble BASIC, you need a 12K machine.

The BASIC source files include

Name	Component
BASIC.PA	Editor Source
BCOMP.PA	Compiler Source
BLOAD.PA	Loader Source
BRTS.PA	Run-time System Source

1. To assemble the editor:

```
.R PAL8
*DEV:BASIC.BN<DEV:BASIC.PA
```

2. To assemble the compiler:

```
.R PAL8
*DEV:BLOAD.BN<DEV:BLOAD.PA
```

3. To assemble the loader:

```
.R PAL8
*DEV:BCOMP.BN<DEV:BCOMP.PA
```

4. The run-time system source is conditionalized for PDP-8/E with EAE. Assembly instructions for each of the supported configurations follow.

To assemble for PDP-12, PDP-8, PDP-8/I or PDP-8/L, or PDP-8E without EAE, type the following command:

```
.R PAL8
*DEV:BRTS.BN DEV:BRTS.PA/W
```

CHAPTER 5

LAB8/E FUNCTIONS FOR OS/8 BASIC

The addition of LAB8/E functions to OS/8 BASIC enables the user to solve a range of real-time and pseudo-real-time problems using a higher-level language. The benefits of approaching real-time problems using BASIC are numerous: a novice programmer can solve problems with little or no assembly language expertise; and in general, the programming effort required for specific problems is dramatically reduced.

The approach taken for specifying each function was to maximize functional flexibility rather than to stress simplicity. Slaving the computer to external events is accomplished by recognizing Schmitt trigger firings. One of the design goals for the LAB8/E functions was to utilize memory efficiently for single precision and displayable data arrays. Another design goal was to incorporate a masking ability for the recognition of bit patterns when reading digital data. This feature allows easy conversion of decimal data into floating-point format when data is received from decimal devices interfaced to the LAB8/E's digital input registers (DR8-E's).

5.1 GENERAL DESCRIPTION

This program contains a set of 12 functions which enable a user of OS/8 BASIC to utilize the following peripherals on a LAB8/E: A/D converter, VC8-E display control, DK8-ES real-time clock, and DR8-EA 12-channel buffered digital I/O. All functions, contained in an overlay called BASIC.UF, reside in the overlay area of BASIC (3400-4577), with the understanding that the entire set of functions is in core whenever a given function is in use. Each function is called by a suitable three-character name, followed by any necessary arguments.

General regulations on arguments passed by the user functions in this package:

- All arguments must be within the following range:

  0<ARGUMENT<4095

  Hence, negative arguments (<0) will cause a fatal error, FM; and positive arguments greater than 4095 (>4095) will cause the fatal error, FO. Fatal errors terminate program execution and return the user to command mode.

- Additional restrictions to arguments will be stated, along with the discussion of each function, later on. Argument errors due to these added restrictions will cause the fatal error, IA (illegal argument).

5.2 PREPARING BASIC FOR LAB8/E FUNCTIONS

The Basic Run-Time System (BRTS) provides for one overlay area and divides a set of infrequently used functions into three separate overlays; namely, BASIC.AF, BASIC.SF and BASIC.FF. Since a logical need for user-written assembly language subroutines exists, a last overlay, BASIC.UF was reserved. It is this last overlay that contains the 12 functions for LAB8/E support. Since the subroutines of this last overlay are determined apart from BRTS, it is necessary that BRTS be given a list of core addresses for each of the user subroutines. It is critical that these links or addresses be specified in the same order that the UDEF statements will appear in the program that calls the functions.

Before writing any program using these functions, it is absolutely necessary to modify BRTS. The following example illustrates how to do this. Notice that in the test programs at the end, the order in the UDEF statements is the same as the ordering of the addresses here. A list of the names of the functions associated with each address is specified to the right for the sake of clarity only.

```
.GET SYS BRTS.SV
.OD
1/**** 5402              used for interrupts
00002 /**** 4456
1560/**** 3400           INI
01561/**** 3454          PLY
01562/**** 3473          DLY
01563 /**** 3600         DIS
01564 /**** 4000         SAM
01565 /**** 4100         CLK
01566 /**** 3541         CLW
01567 /**** 3521         ADC
01570 /**** 4400         GET
01571 /**** 4432         PUT
01572 /**** 4271         DRI
01573 /**** 4313         DRO
^C
.SAVE SYS BRTS.SV
```

Since many of BASIC's functions also reside in overlays, you should take care in using a function that may cause the current set of functions to be overlayed and useful information to be destroyed. For example, the user cannot calculate a set of cosine values and pass them to the PLY function to be stored, because COS resides in BASIC.AF overlay and PLY resides in BASIC.UF.

5.3 DEFINITION OF LAB/8E SUPPORT FUNCTIONS

Once you have modified BRTS to recognize the user function from the BASIC.UF overlay, you may write BASIC programs making use of these functions. If a program requires the use of the Nth function in the ordered list of links, the first (N-1) functions of the list must be defined by UDEF statements or a set of (N-1) dummy-named functions must precede the defining of the Nth function. For example, in

reference to the ordered list of functions in the previous section, if the ADC function is the only one to be used in a particular BASIC program, the UDEF statements must be:

```
                .
                .
                .
10  UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
11  UDEF SAM(C,N,F,T),CLK(R,O,S),CLW(N),ADC(N)
                .
                .
                .
```

or

```
                .
                .
                .
10  UDEF DUA(N),DUB(N),DUC(N),DUD(N)
11  UDEF DUE(N),DUF(N),DUG(N),ADH(N)
                .
                .
                .
```

However, in order to keep careless omissions to a minimum, you should always use the complete set of UDEF's each time you require one or more functions in a program.

INI(N)

The initialize function has a twofold purpose. Its main purpose is to locate the address of the array specified by BASIC's USE statement and retain that address until BASIC.UF is overlayed by one of the other three overlays.

A secondary purpose is to set a pointer to the first location of the array. Consequently, you may use an array to store one set of data followed immediately by a second set of data, provided you call the INI function once only. This means that displayable data (10 bits), and fixed-point data (12 bits) may share the user array at the user's discretion. If, however, you again specify the INI function at the end of the first data run, you cause the first set of data to be overwritten by the second set of data. Hence, INI effectively zeros the array in this case. Whenever you want to use an array in conjunction with one or more of the functions in the BASIC.UF overlay, first dimension the array and then eventually employ the USE statement before the INI function can have meaning. For example:

```
DIM A(3)
   .
   .
   .
USE A
   .
   .
   .
X=INI(0)
   .
   .
   .
```

The argument N, for INI, is a dummy argument and may be any integer; 0, 1, 2, ...

Whenever the functions PLY, DIS, SAM, GET, and PUT are used, make sure that you have called the INI function at least once. When an array is given the dimension N, BASIC allocates (N+1) floating-point words of memory which is actually 3(N+1) single-memory locations. Thus, in the example above, BASIC allocates 4 floating-point words or 12 single-memory locations for the array. Each data value deposited into the user's array by the user functions is a single-precision value (uses one memory word).

PLY(Y)

The purpose of the plot function is to enable a BASIC program to create y-data values and enter them into the user array seguentially, beginning with the first unused location of the array. Each floating-point value is fixed to a 10-bit single-precision value before it is put into the array. The range of the y-data values must be:

$0<y<1.0$

This is easily accomplished by inserting a scaling factor. (Refer to line numbers 26 and 64 of the example program TEST0A.PG at the end of this chapter.)

The data in the user array can be displayed as it is being passed to the array (see DLY function) and/or be refreshed continuously once all values have been entered into the array (see DIS function).

DLY(N)

The delay function is used only in conjunction with the PLY function. It causes the scope to be refreshed with the contents of the user array after each point is processed, so that the graphical progress of data can be observed.

N is an integer such that $1<N<1024$. It specifies the maximum number of points to be eventually displayed. Implied here is the fact that the display will contain only the first N points even if the arrays contain more than N points.

DIS(S,E,N,X)

You use the display function to set up parameters for the displaying of y-data stored in the user array. The display will begin with the desired starting point, S, of the array and display every Nth point while not exceeding the desired endpoint, E (where N = 1, 2, 3,...).

Depending on the value of X, the DIS function has two separate operations:

Operation when X equals zero (X=0): Indication is given to the user overlay functions that a SAM function will be the next BASIC instruction. Consequently the parameters mentioned above are set up so that exactly one of the sampled channels can be displayed "on the

fly". To understand the use of the arguments S,E,N,X, it is necessary to know how the A/D data is stored in the user array. For example, assume 100 samples/channel in each case:

Array	Case 1 SAM CH#0	Case 2 SAM CH#3,4,5
WD1	CH#0	CH#3
WD2	CH#0	CH#4
WD3	CH#0	CH#5
WD4	CH#0	CH#3
WD5	CH#0	CH#4
WD6	CH#0	CH#5
.	.	.
.	.	.
.	.	.
WD100	CH#0	CH#3

To display CASE1, once sampling begins:

```
DIS(1,100,1,0)
```

To display CH#4 of CASE2, once sampling begins:

```
DIS(2,100,3,0)
```

Operation when X is greater than zero (X>0): A user array of y-data is to be displayed immediately. The display is continually refreshed (no return to BASIC) until the operator types CTRL/N on the keyboard.

Displayable y-data values are assumed to be 10-bit single-precision data words.

The x-coordinate for each y-data value is determined by a DELTAX value found as follows:

```
DELTAX = 1023/ [E-S)/N]
```

Due to the outcome of DELTAX, the display may not always use the full width of the scope. However, the display is always centered.

S>1; E>S; (E-S)/N<1023. At least one point must be displayed, and no more than 1024 points may be displayed.

SAM(C,N,P,T)

The sample function is used solely to set up parameters for subsequent sampling of the ADC's or for subsequent sampling of digital input registers (0,1,2), depending on the value of T.

TASK 1 (T=0): Sample the ADC's.

C = First channel # to be sampled; 0<C<17(8).

N = Number of consecutive channels to sample; 1<N<(20(8)-C).

P = Number of sample points/channel; P=0.

TASK 2 (T=0): Sample digital input registers.

C = First register # to be sampled; 0<C<2.

N = Number of consecutive input registers to sample; 1<N<(3-C).

P = Number of samples/register; P=0.

Anytime a SAM instruction is used to sample the ADC's, exactly one channel must be displayed on the fly. However, the sampling rate is not slowed down by this requirement. Hence a DIS function call must precede a SAM function call whenever TASK 1 is chosen.

It is possible to display digital input data so long as only the least significant 10 bits will be displayed. However, this data cannot be displayed on the fly and can only be displayed via the DIS function once all data is in the array.

CLK(R,O,S)

The clock function sets up the clock to be used for A/D sampling, for digital input sampling, or as a simple timing device.

R (rate) = desired frequency at which to run the clock

Value of R	Frequency
1	External input
2	100 HZ
3	1K HZ
4	10K HZ
5	100K HZ
6	1M HZ

O (overflow CNT) = number of clock ticks per interrupt with the clock running at the desired frequency, R. 0<O<4095.

S (Schmitt trigger) (S=0) = Activate all Schmitt triggers and start the clock when any one of the three Schmitt triggers fires. (S=0) Do not activate any Schmitt triggers and start up the clock immediately.

As mentioned above, this single clock function is used to set the clock for one of three separate tasks.

TASK1: Sample the ADC's.

The interrupts are turned on and the program waits in the display loop for a clock overflow, at which time the A/D channel(s) is (are) sampled. The display loop will display the data for the channel specified by the user in the DIS function. When all channels have been sampled the requested number of times, the CLK function returns to BASIC.

When interrupts are turned on, the only possible valid interrupts can be caused by the keyboard or the clock. Hence, any other interrupt is an uncontrollable, spurious interrupt (faulty hardware) that will cause a HLT at location 4466. If this happens, do the following:

1. Set SWITCH REGISTER to 4476 and press ADDR LOAD.

2. Press the CLEAR and CONT switches to return to BASIC.

3. Type CTRL/C to return to the OS/8 Monitor.

TASK2: Sample digital input registers.

At each clock overflow, the digital input register(s) is (are) sampled. When all registers have been sampled the requested number of times, the CLK function returns to BASIC.

NOTE

The sampled data from the ADC's or the digital input registers is stored sequentially in the user's array.

TASK3: A simple timing device.

The clock is set up and started (unless it is to be started when a Schmitt trigger fires) and then returns to BASIC.

The following illustrates what sequence of instructions are needed for each task.

```
TASK1              TASK2              TASK3

   .                  .                  .
   .                  .                  .
   .                  .                  .
DIM A(n)           DIM A(n)           Z=CLK(R,O,S)
USE A              USE A                 .
   .                  .                  .
   .                  .                  .
   .                  .
W=INI(0)           W=INI(0)
X=DIS(C,N,P,T)     Y=SAM(C,N,P,1)
Y=SAM(C,N,P,0)     Z=CLK(R,O,S)
Z=CLK(R,O,S)          .
   .                  .
   .                  .
   .
```

CLW(N)

After the clock has been set up by CLK as a simple timer, this clock wait function, when called, simply returns to BASIC whenever a clock overflow occurs, and/or whenever a Schmitt trigger fires, provided S was a non-zero argument in CLK.

Upon return to BASIC, a number is returned to the caller indicating whether the return was due to a clock overflow, a Schmitt trigger, or a clock overflow and the firing of a Schmitt trigger simultaneously. The number also indicates whether one of the above conditions occurred before or after the CLW function was called. N is a dummy argument (N=0,1,2,...).

The following table illustrates the various numbers returned.

Case 1: Clock overflowed or a Schmitt trigger fired after CLW is called.

Overflow Only	Schmitt Trigger Only	Simultaneously
0	1 (Trigger 1 fired)	-1
	2 (Trigger 2 fired)	-2
	3 (Trigger 1 & 2 fired)	-3
	4 (Trigger 4 fired)	-4
	5 (Trigger 1 & 4 fired)	-5
	6 (Trigger 2 & 4 fired)	-6
	7 (Trigger 1,2 & 4 fired)	-7

Case 2: Clock overflowed or a Schmitt trigger fired before CLW is called.

Overflow Only	Schmitt Trigger Only	Simultaneously
-8	9	-9
	10	-10
	11	-11
	12	-12
	13	-13
	14	-14
	15	-15

The TEST4A.PG and TEST5A.PG examples make use of the CLW function.

The CLW function has many useful applications. For example, you may time subroutines by starting the clock with a specific rate and overflow count. After you call the subroutine and the subroutine is completed, call the CLW function to see if an immediate return is obtained. This timing is empirical in that you would keep changing the rate and/or overflow count until Case 2 occurred. As a second example, you may use Schmitt trigger firing to branch to a particular subroutine or to notify the program to proceed with specific tasks such as reading digital data or sampling an analog input. Thirdly, time-interval histograms and post-stimulus histograms are also possible (see TST20A.PG).

ADC(N)

This function is issued any time you wish to sample A/D channel N. The 10-bit data value is floated and returned to the caller for immediate examination. 0<N<17(8).

The BASIC statement W=ADC(3) asks that A/D channel #3 be sampled and the floating-point value be assigned to W.

The TEST5A.PG example illustrates one use of the ADC function.

GET(M,L)

You use this function to get one 12-bit word from the user array, mask out certain bits, and return the result as a floating-point number to the caller.

L is Lth location of the user array. Hence, if an array has N single-precision words, L can take on meaningful values of 1,2,3,...,N.

NOTE

Although BASIC allows 0 to be a meaningful value in a dimension statement such as DIM A(0), you must remember that L always begins with 1, where 1 stands for the first single-word location of the array. Thus DIM A(0) specifies an array of one floating-point word (three one-word locations).

M is a masking number such that 0<M<4095. This floating-point number is converted to a 12-bit binary number between 0 and 7777. Those bits that are zero will mask out or eliminate those bits in the array value. If M=0, then no masking is done and the 12 bit array value is returned intact. M=0 and M=4095 have the same meaning.

The BASIC statement Y=GET(15,2) gets the second word of the user array, masks out all bits except bits 8,9,10,11, and assigns the floating-point result to Y. Consequently, if an array is as follows:

```
single prec WD1     5678
single prec WD2     1234     Fl. pt. word 0
single prec WD3     4455
                      .
                      .
                      .
WD2 = 1234(8) = 001010011100(2)
MASK = 15(10) = 17(8) - 000000001111(2)
The 12 bit value after masking is:
                        000000001100(2) = 12(10)
                Hence, Y=12
```

PUT(M,L)

This function enables a floating-point number to be fixed to a single 12-bit word and put into the user's array.

L is Lth location of the user's array. For an array of N single-precision words, L can take on meaningful values of 1,2,3,...,N.

M is the floating-point number to be fixed and stored in the array. 0<M<4095.

NOTE

> Both GET and PUT functions imply that a user's array must not exceed 4096 memory locations, because of the general restriction on any argument for these user functions.

The BASIC statement Y=PUT(128,4) means fix 128 to 12 bits (000 010 000 000(2)) and put the value into the fourth word of the user array. TST15A.PG, TST16A.PG, TST17A.PG and TST18A.PG illustrate the use of functions GET and PUT.

DRI(N)

This function is issued any time you wish to sample a digital input register, N (0<N<2). The 12-bit digital value is returned to the user as a floating-point number. Basic statement: X=DRI(0) means that input register #0 is sampled and the floating-point result is assigned to X.

DRO(M,N)

This function is issued any time you wish to set the bits of a digital output register, N(0<N<2). The output register bits are set via the value of M (1<M<4095). If M=0, the output register is cleared; otherwise the bits of the register remain set. Hence, additional bits of the register can be set while maintaining those set earlier.

Basic statement: Z=DRO(9,1) means set bits 8 and 11 of output register #1 if not already set.

9(10) - 000000001001(2)

TST13A.PG and TST15A.PG illustrate the use of the DRI and DRO functions.

5.4 LAB8/E EXAMPLES

The following set of BASIC programs illustrates a number of ways the user functions may be implemented. Each program has been kept as simple as possible.

Note that for TST12A.PG, TST13A.PG and TST15A.PG a battery-powered black box was used to interact with the digital I/O registers. The box contained a set of 12 switches which could set any combination of bits for the digital input register; it also contains a row of 12 lights lighted by the contents of the 12-bit digital output register. When running TST18A.PG, use the data from TST17A.PG.

```
1 REM -           PROGRAM NAME:  TESTOA.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
9 REM -
10 REM - CALC 1024 PTS & DISPLAY ON FLY.
11 REM - WHEN DONE DISPLAY EVERY 10TH PT.
12 REM -
20 USE A
22 Z=INI(0)
24 FOR N=1 TO 1024
26 Y=(3*N-2)/3071
28 X=PLY(Y)
30 W=DLY(1024)
32 NEXT N
34 V=DIS(1,1024,10,1)
49 REM -
50 REM - CALC 30 PTS & DISPLAY ONLY
51 REM - WHEN DONE.
60 Z=INI(0)
62 FOR N=1 TO 30
64 Y=(2+N+1)/61.1
66 Z=PLY(Y)
68 NEXT N
70 V=DIS(1,30,1,1)
80 END
```

```
1 REM -           PROGRAM NAME:  TEST1A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
10 REM -
11 REM - SAMPLE CHAN 0 (1024 TIMES);DISPLAY
12 REM - ALL PTS ON THE FLY.
13 REM - 10 INTERRUPTS/SEC
```

```
14 REM -
20 USE A
21 W=INI(0)
22 W=DIS(1,1024,1,0)
24 X=SAM(0,1,1024,0)
26 Y=CLK(3,100,0)
28 Z=DIS(1,1024,1,1)
40 REM -
41 REM - SAMPLE CHANNELS 0,1 (100 TIMES EACH).
42 REM - 10 INTERRUPTS/SEC;DISPLAY CHAN 0 WHILE
43 REM - SAMPLING,WHEN DONE SHOW THREE DIFF
44 REM - DISPLAYS: DISPLAY CHAN 0--HIT ^N DISPLAY
45 REM - CHAN 1--HIT ^N DISPLAY CHAN 0&1.
50 USE A
51 W=INI(0)
52 W=DIS(1,200,2,0)
54 X=SAM(0,2,100,0)
56 Y=CLK(3,100,0)
58 Z=DIS(1,200,2,1)
60 U=DIS(2,200,2,1)
62 V=DIS(1,200,1,1)
70 END
```

```
1 REM -          PROGRAM NAME:  TEST2A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
10 REM -
11 REM - CALC A PARABOLA OF 601 PTS AND DISPLAY
12 REM - ON THE FLY. WHEN DONE DISPLAY EVERY 10TH
13 REM - PT OF PARABOLA.
14 REM -
20 USE A
22 Z=INI(0)
24 FOR N=-300 TO 300
26 Y=(N*N)/100000
28 X=PLY(Y)
30 W=DLY(601)
32 NEXT N
34 V=DIS(1,601,10,1)
50 REM -
51 REM - CALC A CUBIC OF 601 PTS & DISPLAY ON FLY
52 REM - WHEN DONE DISPLAY EVERY 10TH PT.
53 REM -
60 Z=INI(0)
62 FOR N=-300 TO 300
64 Y=(N*N*N+27000000)/54000010
66 X=PLY(Y)
68 W=DLY(601)
70 NEXT N
72 V=DIS(1,601,10,1)
80 END
```

```
1 REM -           PROGRAM NAME:  TEST3A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
10 REM -
11 REM - ILLUSTRATE ABILITY TO ACCESS USER BUFFER.
12 REM - PUT NUMBERS 1-10 INTO BUF IN THAT ORDER.
13 REM - & READ THEM OUT IN THE REVERSE ORDER.
14 REM -
20 Z=INI(0)
22 FOR N=1 TO 10
24 PRINT N
26 T=N
28 R=PUT(T,N)
30 NEXT N
32 FOR N=1 TO 10
34 N=11-N
36 P=GET(0,M)
38 PRINT P
40 NEXT N
50 END
```

```
1 REM -           PROGRAM NAME:  TEST4A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 REM - SAMPLE CHAN 0 IF CLOCK O.F.
7 REM - SAMPLE CHAN 1 IF SCHMITT ONLY
8 REM - SAMPLE CHAN IF BOTH FIRE
9 REM - IF EARLY, TELL USER
10 REM - ROUTINE ALSO OUTPUTS Z
11 X=CLK(3,4000,1)
12 FOR N=1 TO 10
14 Z=CLW(0)
15 PRINT "Z=";Z
16 IF Z=0 GOTO 30
18 IF Z<0 GOTO 24
19 IF Z<8 GOTO 34
20 IF Z=8 GOTO 40
21 GOTO 40
24 IF Z<-8 GOTO 40
26 W=ADC(2)
28 GOTO 36
30 W=ADC(0)
31 GOTO 36
34 W=ADC(1)
36 PRINT W
37 GOTO 42
40 PRINT "EARLY"
42 NEXT N
50 END
```

```
1 REM -          PROGRAM NAME:  TEST5A.PG
2 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
3 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
4 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
5 DIM A(342)
10 REM -
11 REM - USE CLK AS A SIMPLE TIMER.
12 REM - SAMPLE CHAN 0 EVERY 4TH SEC AND PUT VAL TO TTY
13 REM - DO THIS 10 TIMES
14 REM -
20 X=CLK(3,4000,0)
22 FOR I=1 TO 10
24 Y=CLW(0)
26 Z=ADC(0)
28 PRINT Z
30 NEXT I
40 REM -
41 REM - USE CLK AS A SIMPLE TIMER
42 REM - SAMPLE CHAN 1 TEN TIMES & SYNC OFF ANY
43 REM - SCHMITT TRIGGER
44 REM -
50 X=CLK(4,4000,1)
52 FOR I=1 TO 10
54 Y=CLW(0)
56 Z=ADC(0)
58 PRINT Z
60 NEXT I
70 END
```

```
1 REM -          PROGRAM NAME: TEST7A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
7 USE A
8 REM - DISPLAY A TRIANGLE
10 Z=INI(0)
12 FOR N=1 TO 30
14 Y=N/30.1
16 W=PLY(Y)
18 Z=1/30.1
20 U=PLY(Z)
22 P=DLY(118)
24 NEXT N
26 FOR N=1 TO 29
27 M=30-N
28 Y=N/30.1
30 W=PLY(Y)
32 Z=1/30.1
34 U=PLY(Z)
36 P=DLY(118)
38 NEXT N
40 V=DIS(1,118,1,1)
42 END
```

```
1 REM -           PROGRAM NAME:  TEST8A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
6 DIM A(342)
10 REM -
11 REM - SAMPLE CHAN 0 100 TIMES;DISPLAY;
12 REM - HOWEVER SYNC OFF SCHMITT TRIGS.
14 REM -
32 USE A
34 W=INI(0)
36 W=DIS(1,100,1,0)
38 X=SAM(0,1,100,0)
40 Y=CLK(3,100,1)
42 Z=DIS(1,100,1,1)
50 END
```

```
1 REM -           PROGRAM NAME:  TEST9A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
10 REM -
11 REM - CALC A PARABOLA OF 401 PTS AND DISPLAY ON FLY
13 REM -
20 USE A
22 Z=INI(0)
24 FOR N=-2 TO 200
26 Y=(N*N)/4000!
28 X=PLY(Y)
30 W=DLY(401)
32 NEXT N
50 REM -
51 REM - CALC A CUBIC OF 401 PTS & DISPLAY ON FLY
52 REM - SHOW PARABOLA. WHEN DONE DISPLAY EVERY PT
53 REM - & THEN EVERY 10TH PT
54 REM -
62 FOR N=-200 TO 200
64 Y=(N*N*N+8000000)/16000010
66 X=PLY(Y)
68 W=DLY(802)
70 NEXT N
72 V=DIS(1,802,1,1)
74 V=DIS(1,802,10,1)
80 END
```

```
1 REM -           PROGRAM NAME:  TST10A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
7 REM - THIS ROUTN RETURNS 4 DIGITS-3BITS/DIGIT
10 USE A
11 Z=INI(0)
12 PRINT "VALUE"
14 INPUT Y
16 Z=PUT(Y,1)
18 P=GET(7,1)
```

```
19 PRINT P
20 P=GET(56,1)
21 PRINT P
22 P=GET(448,1)
23 PRINT P
24 P=GET(3584,1)
25 PRINT P
26 GOTO 12
30 END
```

```
1 REM -          PROGRAM NAME:  TST12A.PG
2 REM -
3 REM - THIS ROUTN SAMPLES DIGITAL BOARD
4 REM - #1 TEN TIMES, ONCE EVERY 4 SECS & PUTS
5 REM - THE VALUES INTO USER BUF THEN IT PRINTS
6 REM - OUT THE 10 VALUES
10 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
11 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
12 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
20 DIM A(342)
22 USE A
23 W=INI(0)
24 X=SAM(1,1,10,1)
26 Y=CLK(3,4000,0)
28 FOR N=1 TO 10
30 W=GET(0,N)
32 PRINT W
34 NEXT N
40 END
```

```
1 REM -          PROGRAM NAME:  TST13A.PG
2 REM -
3 REM - TEST THE OUTPUT REG-SEE THE LIGHTS LITE
4 REM - UP . OCTAL INPUT LIGHTS THE LIGHTS AND
5 REM - THE LAMP? AN INPUT OF 0 CLEARS THE OUTPUT REG
10 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
11 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
12 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
14 W=DRO(0,1)
16 PRINT "NUMBER"
18 INPUT Y
19 IF Y=0 GOTO 14
20 W=DRO(Y,1)
22 GOTO 16
30 END
```

```
1 REM -          PROGRAM NAME:   TST15A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM (C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(342)
7 REM - THIS ROUTN RETURNS 3 DIGITS-4 BITS/DIGIT
8 REM - (MASKING) IT FIRST OUTPUTS THE DECIMAL
9 REM - EQUIV OF THE NUMBER
10 USE A
11 Z=INI(0)
12 W=DRI(1)
```

```
13 PRINT W
16 X=PUT(W,1)
18 P=GET(15,1)
19 PRINT P
20 P=GET(240,1)
21 PRINT P
22 P=GET(3840,1)
23 PRINT P
24 PRINT "WASTE TIME"
25 INPUT R
26 GOTO 12
30 END
```

```
1 REM -           PROGRAM NAME:   TST16A.
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(3)
7 REM - THIS ROUTN SHOWS THAT ANY N;0<=N<=4095
8 REM - PUT INTO A USER BUF IS RETURNED AS THE
9 REM - SAME VALUE.
10 USE A
11 Z=INI(0)
12 PRINT "NUMBER"
14 INPUT Y
16 X=PUT(Y,1)
18 Z=GET(0,1)
20 PRINT Z
26 GOTO 12
30 END
```

```
1 REM -           PROGRAM NAME:   TST17A.PG
2 REM - FILL AN ARRAY OF 30 WORDS WITH THE
3 REM - FIRST 30 INTEGERS. WRITE THE ARRAY
4 REM - OUT TO DECTAPE.
5 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
6 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
7 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
8 DIM A(9)
9 USE A
10 X=INI(0)
11 FOR N=1 TO 30
12 PRINT N
13 X=PUT(N,N)
14 NEXT N
16 FILEVN#1:"DTA1:DATA.PG"
22 FOR I=0 TO 9
24 PRINT #1:A(1)
26 NEXT I
28 CLOSE #1
30 END
```

```
1 REM -          PROGRAM NAME:   TST18A.PG
2 REM - READ INTO AN ARRAY 10 FL PT WDS
3 REM - (30 INTEGERS FROM MS) WRITE OUT THE
4 REM - 30 INTEGERS ON TTY
5 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
6 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
7 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
8 DIM A(9)
9 USE A
20 FILEN #1:"DTA1:DATA.PG"
22 FOR I=0 TO 9
24 INPUT #1:A(I)
26 NEXT I
28 CLOSE #1
29 X=INI(0)
30 FOR N=1 TO 30
32 X=GET(0,N)
34 PRINT X
36 NEXT N
40 END
```

```
1 REM -          PROGRAM NAME:   TST19A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,P,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
6 DIM A(16)
10 REM - SAMPLE CHAN 0 50 TIMES;SYNC OFF SCHMITT;
11 REM - 10 INTERRUPTS/SEC;WHEN DONE DISPLAY TILL ^N;
12 REM - THEN WRITE OUT DATA TO DTA1;
20 USE A
21 W=INI(0)
22 W=DIS(1,50,1,0)
24 X=SAM(0,1,50,0)
26 Y=CLK(3,104,1)
28 Z=DIS(1,50,1,1)
29 FILEVN #1:"DTA1:SAM.DA"
30 FOR I=0 TO 16
32 PRINT #1:A(I)
34 NEXT I
36 CLOSE #1
38 REM - DISPLAY A PARABOLA
40 P=INI(0)
42 FOR N=-25 TO 25
44 Y=(N*N)/625.1
46 X=PLY(Y)
48 W=DLY(51)
50 NEXT N
52 V=DIS(1,51,1,1)
54 REM - READ DATA BACK IN & DISPLAY IT AS BEFORE
56 FILEN #1:"DTA1:SAM.DA"
58 FOR I=0 TO 16
60 INPUT #1:A(I)
62 NEXT I
64 W=INI(0)
66 Z=DIS(1,50,1,1)
68 END
```

```
1 REM -           PROGRAM NAME:   TST20A.PG
2 REM -
3 UDEF INI(N),PLY(Y),DLY(N),DIS(S,E,N,X)
4 UDEF SAM(C,N,F,T),CLK(R,O,S),CLW(N),ADC(N)
5 UDEF GET(M,L),PUT(M,L),DRI(N),DRO(M,N)
10 DIM X(100),Y(100),A(67)
11 REM - J1=BINS IN LATENCY(#EPOCHS TILL DONE)
12 REM - T1=BIN WIDTH(TIH) IN MS(#MS/CLK O.F.)
13 REM - T2=BIN WIDTH OF LATENCY(#CLK O.F./EPOCHS)
16 PRINT "J1,T1,T2?"
18 INPUT J1,T1,T2
20 I=0
21 J=0
22 K=0
23 Y=CLK(3,T1,1)
25 Z=CLW(0)
30 IF Z=0 GOTO 100
32 IF Z<0 GOTO 36
34 IF Z<8 GOTO 200
35 GOTO  38
36 IF Z>-8 GOTO 200
37 REM - INCR UNDERFLO BIN 0
38 I=0
39 GOTO 300
99 REM - CLK O.F. ONLY;BMP HIST BIN
100 I=I+1
102 IF I<>100 GOTO 110
103 REM - END OF TIME,BMP HIST BIN
104 X(100)=X(100)+1
105 I=0
109 REM - BMP LATENCY CTR
110 K=K+1
112 IF K<>T2 GOTO 25
113 REM - AN EPOCH IS DONE
114 K=0
116 J=J+1
118 IF J=J1 GOTO 500
119 REM - MORE EPOCHS TO GO?
120 GOTO 25
199 REM - CLK O.F. AND SCHMITT TRIG
200 X(I)=X(I)+1
202 Y(J)=Y(J)+1
204 GOTO 100
299 REM - SCHMITT TRIG ONLY
300 X(I)=X(I)+1
302 Y(J)=Y(J)+1
304 GOTO 25
498 REM - GET LARGEST BIN VALUE TO BE USED AS A
499 REM - SCALE FACTOR FOR DISPLAY
500 USE A
503 Q=0
504 FOR I=0 TO 100
506 Z=X(I)
508 IF Q>=Z GOTO 516
510 Q=Z
516 NEXT I
549 REM - SCALE ALL BIN VALUES FOR MAX DISPLAY
550 W=INI(0)
551 FOR I=0 TO 100
552 Z=X(I)
554 Y=Z/(Q+1)
555 W=PLY(Y)
556 NEXT I
598 REM - GET LARGEST LATENCY VAL TO BE
```

```
599 REM - USED AS A SCALE FACTOR FOR DISPLAY
600 Q=0
602 FOR I=0 TO 100
604 Z=Y(I)
606 IF Q>=Z GOTO 610
608 Q=Z
610 NEXT I
699 REM - SCALE ALL LATENCY VALS FOR MAX DISPLAY
700 FOR I=0 TO 100
702 Z=Y(I)
704 Y=Z/(Q+1)
706 W=PLY(Y)
708 NEXT I
710 REM - DISPLAY 'TIH'
711 V=DIS(1,101,1,1)
712 REM - DISPLAY LATENCY
720 V=DIS(102,202,1,1)
725 REM - DISPLAY BOTH 'TIH' & LATENCY SIDE BY SIDE
726 V=DIS(1,202,1,1)
800 END
```

5.5 GETTING ON THE AIR WITH BASIC

DECtape users:

Transfer the user overlays, BASIC.UF, from the DECtape provided with the software kit to the OS/8 system device.

```
.R PIP
*SYS:BASIC.UF<DTAn:BASIC.UF/I          (where n=0,1,2,...,7)
*^C
```

Papertape users:

Use the ABSLDR to read into core the user overlays that are in binary format on the paper tape, provided with the software kit. Then create a save file on the system device.

```
.R ABSLDR
*PTR:$^          (where $ symbolizes striking the ALTMODE key)
.SAVE SYS BASIC.UF 3400-4577
```

5.6 LAB8/E FUNCTION SUMMARY

Table 5-1
LAB8/E Function Summary

Function	Explanation
INI(N)	Locate the address of the user array and initialize a pointer to start of the array. N is a dummy argument.
PLY(Y)	Y-data created via the BASIC program is deposited into the user array sequentially. 0<Y<.0

(continued on next page)

Table 5-1 (Cont.)
LAB8/E Function Summary

Function	Explanation
DLY(N)	Used in conjunction with PLY, the scope is refreshed with the contents of the user array after each point is processed. 1<N<1024 and N specifies the maximum number of points to be eventually displayed.
DIS(S,E,N,X)	Meaning #1 (X=0). Set up parameters to display ADC data once sampling begins. Meaning #2 (X=0). An array of y-data is to be displayed immediately. In both cases, the display begins with point S of the array, and every Nth point is displayed while not exceeding the desired point E.
SAM(C,N,P,T)	Used to set up parameters for subsequent sampling of the ADC's (T=0) or sampling of digital input registers (T=0). C is the first channel # or digital input register #. N is the number of consecutive channels or registers to sample. P is the number of samples per channel or register.
CLK(R,O,S)	Set up the clock for A/D sampling, digital input sampling or for use as a simple timer. R is the desired rate, O is the overflow count, and S activates the Schmitt triggers.
CLW(N)	This function returns to the caller a number, indicating whether the clock overflowed or a Schmitt trigger fired and whether these occurred before or after CLW was called.
ADC(N)	This function is issued any time the user wishes to sample A/D channel N.
GET(M,L)	A 12-bit number from the user array at location L is masked with the number M and returned to the caller.
PUT(M,L)	A floating-point number, M, is fixed to 12 bits and stored in the user array at location L.
DRN(N)	This function is used any time the user wishes to sample a digital input register N.
DRO(M,N)	The bits of digital output register N are set via the value of M.

APPENDIX A

SUMMARY OF BASIC EDITOR COMMANDS

Command	Function
BYE	Exits from the editor and returns control to the monitor
LIst	Displays the program statements in the workspace with a header
LISTNH	Displays the program statements in the workspace, without a header
NAme	Renames the program in the workspace
NEw	Clears the workspace and tells the editor the name of the program the user is about to type
OLd	Clears the workspace, finds a program on the disk, and puts in into the workspace
RUn	Executes the program in the workspace, after displaying a header
RUNNH	Executes the program in the workspace, without displaying a header
SAve	Puts the program in the workspace on a disk
SCratch	Erases all statements from the workspace

APPENDIX B

SUMMARY OF BASIC STATEMENTS

Statement	Function
CHAIN	Executes another program
	Example: 40 CHAIN "SYS:PROG.BA"
CLOSE#	Closes a file
	Example: 100 CLOSE#1
DATA	Sets up a list of values to be used by the READ statement
	Example: 240 DATA "FIRST",2,3
DEF	Defines functions
	Example: 10 DEF FND(S)=S+5
DIM	Describes a string and/or any subscripted variables
	Example: 50 DIM B(3,5),D$(3,72)
END	Terminates program compilation and execution
	Example: 100 END
FILE#	Defines and opens a file
	Example: 20 FILEVN#2:"RXA1:DATA.NV"
FOR-TO-STEP	Describes program loops (used with NEXT)
	Example: 60 FOR X=1 TO 10 STEP 2
GOSUB	Transfers control to a subroutine (used with RETURN)
	Example: 50 GOSUB 100
GOTO	Transfers control to another statement
	Example: 100 GOTO 50

IF	Tests the relationship between two variables, numbers, or expressions Example: 20 IF A=0 THEN 50
IF END#	Tests for the end of a string file Example: 60 IF END#3 THEN 100
INPUT	Accepts data from the terminal Example: 80 INPUT A,B,C
INPUT#	Reads data from a file Example: 50 INPUT#1:A$
LET	Assigns a value to a variable Example: 90 LET A$="XYZ"
NEXT	Indicates the end of a program loop (used with FOR) Example: 140 NEXT I
PRINT	Displays data on the screen Example: 200 PRINT A,"X";6
PRINT#	Writes data to a file Example: 180 PRINT#1:J
RANDOMIZE	Causes the RND function to produce a different set of numbers each time the program is run Example: 10 RANDOMIZE
READ	Sets variables equal to the values in DATA statements Example: 50 READ A$,B
REM	Inserts comments into the program Example: 30 REM COMPUTE EARNINGS
RESTORE	Sets program READ statements back to the beginning of the DATA list Example: 85 RESTORE
RESTORE#	Resets a file pointer back to the beginning of that file Example: 130 RESTORE#3

RETURN	Returns control from a subroutine (used with GOSUB) Example: 115 RETURN
STOP	Terminates program execution Example: 40 STOP
UDEF	Defines the syntax of a call to a user-coded function
USE	Identifies lists and arrays referenced by a user-coded function

APPENDIX C

SUMMARY OF BASIC FUNCTIONS

Command	Function
ABS(X)	Returns the absolute value of an expression Example: 10 LET X=ABS(-66) will assign X a value of 66
ASC(X$)	Converts a one-character string to its code number Example: 20 PRINT ASC("B") will display 2
ATN(X)	Calculates the angle (in radians) whose tangent is given as the argument Example: 30 LET X=ATN(.57735) will assign X a value of 0.523598
CHR$(X)	Converts a code number to its equivalent character Example: 40 PRINT CHR$(1) will display A
COS(X)	Returns the cosine of an angle specified in radians Example: 50 LET Y=COS(45*3.14159)/180 will assign Y a value of 0.707108
DAT$(X)	Returns the current system date Example: 60 PRINT DAT$(X) will display the system date, such as 07/20/77
EXP(X)	Calculates the value of e raised to a power, where e is equal to 2.71828 Example: 30 IF Y>EXP(1.5) GOTO 70 will go to line 70 if Y is greater than 4.48169
INT(X)	Returns the value of the nearest integer not greater than the argument Example: 60 LET X=INT(34.67) will assign X the value 34

LEN(X$) Returns the number of characters in a string

Example: 10 PRINT LEN ("DOG")

will display 3

LOG(X) Calculates the natural logarithm of the argument

Example: 10 PRINT LOG(959)

will display 6.86589

PNT(X) Outputs nonprinting characters for terminal control

Example: 50 PRINT PNT(13)

will move the cursor to the left margin of the current line

POS(X$,Y$,Z) Returns the location of a specified group of characters (Y$) in a string (X$) starting at a character position (Z)

Example: 60 LET V=POS("ABCDBC","BC",4)

will assign V a value of 5

RND(X) Returns a random number between (but not including) 0 and 1

Example: 70 PRINT RND(X)

will display a decimal number, such as 0.361572

SEG$(X$,Y,Z) Returns the sequence of characters in a string (X$) between two positions in the string (X,Y)

Example: 30 LET R$=SEG$("ABCDEF",2,4)

will assign R$ a value of BCD

SGN(X) Returns 1 if the argument is positive, 0 if it is zero, and -1 if it is negative

Example: 200 PRINT 5*SGN(-6)

will display -5

SIN(X) Returns the sine of an angle specified in radians

Example: 30 LET B=SIN(30*3.14159/180)

will assign B a value of 0.5

SQR(X) Returns the positive square root of an expression

Example: 40 PRINT SQR(16)

will display 4

STR$(X) Converts a number into a string

Example: 120 PRINT STR$(1.76111124)

will display the string 1.76111

TAB(X)	Positions characters on a line Example: 70 PRINT "A";TAB(5);"B" will display A B
TRC(1)	Causes BASIC to display the line number of each statement in the program as it is executed Example: 10 V=TRC(1) will display the line number of each statement executed until a TRC(0) is encountered
VAL(X$)	Converts a string to a number Example: 90 PRINT VAL("2.46lll")*2 will display 4.92222

APPENDIX D

BASIC ERROR MESSAGES

D.1 COMPILER ERROR MESSAGES

The following error messages are generated by the BASIC compiler:

CH	ERROR IN CHAIN STATEMENT
DE	ERROR IN DEF STATEMENT
DI	ERROR IN DIM STATEMENT
FN	ERROR IN FILE NUMBER OR NAME
FP	INCORRECT FOR STATEMENT
FR	ERROR IN FUNCTION ARGS
IF	ERROR IN IF STATEMENT
IC	I/C ERROR
LS	MISSING EQUALS SIGN IN LET
LT	STATEMENT TOO LONG
MD	MULTIPLY DEFINED LINE NUMBER
ME	MISSING END STATEMENT
MO	OPERAND EXPECTED, NOT FOUND
MP	PARENTHESIS ERROR
MT	OPERAND OF MIXED TYPE
NF	NEXT STATEMENT WITHOUT FOR
NM	MISSING LINE NUMBER
OF	OUTPUT FILE ERROR
PD	PUSHDOWN STACK OVERFLOW
QS	STRING LITERAL TOO LONG
SS	BAD SUBSCRIPT OR FUNCTION ARG
ST	SYMBOL TABLE OVERFLOW
SY	SYSTEM INCOMPLETE
TB	PROGRAM TOO BIG
TD	TOO MUCH DATA IN PROGRAM
TS	TOO MANY CHARS IN STRING
UD	ERROR IN UDEF STATEMENT
UF	FOR STATEMENT WITHOUT NEXT
US	UNDEFINED STATEMENT NUMBER
UU	USE STATEMENT ERROR
XC	CHARS AFTER END OF LINE

D.2 RUN-TIME SYSTEM ERROR MESSAGES

The following error messages are generated by the BASIC run-time system:

BO	NO MORE BUFFERS AVAILABLE
CI	IN CHAIN, DEVICE NOT FOUND
CL	IN CHAIN, FILE NOT FOUND
CX	CHAIN ERROR
DA	READING PAST END OF DATA
DE	DEVICE DRIVER ERROR
DC	NO MORE ROOM FOR DRIVERS
DV	ATTEMPT TO DIVIDE BY ZERO
EF	LOGICAL END OF FILE
EM	NEGATIVE NUMBER TO REAL POWER
EN	ENTER ERROR
FB	USING FILE ALREADY IN USE
FC	CLOSE ERROR
FE	FETCH ERROR
FI	CLOSING OR USING UNOPENED FILE
FM	FIXING NEGATIVE NUMBER
FN	ILLEGAL FILE NUMBER
FO	FIXING NUMBER>4095
GR	RETURN WITHOUT GOSUB
GS	TOO MANY NESTED GOSUBS
IA	ILLEGAL ARG IN UDEF
IF	ILLEGAL DEV:FILENAME
IN	INQUIRE FAILURE
IO	TTY INPUT BUFFER OVERFLOW
LM	TAKING LOG OF NEGATIVE NUMBER
OE	DRIVER ERROR WHILE OVERLAYING
OV	NUMERIC OR INPUT OVERFLOW
PA	ILLEGAL ARG IN POS
RE	READING PAST END OF FILE
SC	CONCATENATED STRING TOO LONG
SL	STRING TOO LONG OR UNDEFINED
SR	READING STRING FROM NUMERIC FILE
ST	STRING TRUNCATION ON INPUT
SU	SUBSCRIPT OUT OF RANGE
SW	WRITING STRING INTO NUMERIC FILE
VR	READING VARIABLE LENGTH FILE
WE	WRITING PAST END OF FILE

INDEX

FORTRAN IV

CONTENTS

CONTENTS (Cont.)

CONTENTS (Cont.)

CONTENTS (Cont.)

FIGURES

Page

TABLES

CHAPTER 1

SYSTEM OVERVIEW

OS/8 FORTRAN IV provides full standard ANSI FORTRAN IV under the OS/8 operating system. The FORTRAN IV package requires a minimum hardware environment consisting of a PDP-8 family processor with at least 8K of mainframe memory, a console terminal, and at least 96K of mass storage. The system is automatically self-expanding to employ a KE8-E Extended Arithmetic Element, FPP-12 Floating-Point Processor, up to 32K of mainframe memory, and any bulk storage or peripheral I/O devices that may be present in the system.

Although such factors as maximum program size and minimum execution time depend heavily on the hardware configuration on which any program is run, OS/8 FORTRAN IV affords the full capability of the FORTRAN IV language, even on a minimum configuration, subject only to the restriction that double-precision and complex number operations require an FPP-12 with extended precision option. The system is highly optimized with respect to memory requirements, and an overlay feature is included that permits programs requiring up to 300K of virtual storage to run on a PDP-8 or PDP-12. The library functions permit the user to access a number of laboratory peripherals, to evaluate a number of transcendental functions, to manipulate alphanumeric strings, and to output to a standard incremental plotter.

A FORTRAN IV program written by the user is called a source program, to distinguish it from the various object programs generated by the OS/8 FORTRAN IV system. Source programs may be prepared off line on punched cards or low-speed paper tape; however, it is usually most convenient to prepare source programs on line by means of an editing program such as TECO or EDIT. The source file produced in this manner is an image of the corresponding punched-card file, with carriage return and line feed characters separating adjacent statements (that would otherwise appear on adjacent punched cards) and ASCII spaces or tabs entered in place of blank columns. Because of the close analogy between punched-card source files and other types of source files, the terms "character" and "column" are used interchangeably in this manual.

Once a source program has been prepared, it is supplied as input to the FORTRAN IV compiler, which translates each FORTRAN statement into one or more RALF (Relocatable Assembly Language, Floating-point) statements and produces an output file containing an assembly language version of the source program, plus an optional annotated listing of the source.

This is accomplished in three passes. System program F4.SV begins compilation by building a symbol table and generating intermediate code. F4 chains to PASS2.SV automatically, and PASS2 calls PASS20.SV to complete the translation into assembly language during compilation pass 2. If a source listing was requested, PASS20 chains to PASS3.SV automatically, and PASS3 generates the listing during pass 3. Like PASS2, PASS20 and PASS3 are never accessed directly by the user.

The RALF assembly language output produced by the compiler must be assembled by system program RALF.SV, the RALF assembler. (See Section 1.2 for a description of the RALF assembler.) During assembly, each RALF assembly language statement is translated into one or more instructions for either the PDP-8 processor or the FPP; an output file is then created containing a relocatable binary version of the assembly language input. This is accomplished in two passes; a third pass is executed to generate an annotated listing of the assembly language input file, if requested.

The relocatable binary file produced by the RALF assembler is a machine language version of a single program or subroutine. This file, called a RALF module, must be linked with its main program (if it is a subroutine) and with any other subroutines, including subroutines from the library (e.g., FORLIB.RL) that it requires in order to execute. System program LOAD.SV, the OS/8 FORTRAN IV loader, accepts a list of RALF module specifications from the console terminal and builds a loader image file containing a relocated main program linked to relocated versions of all subroutines and library components that the mainline requires in order to execute.

The loader image file is an executable core load, complete except for run-time I/O specifications. It may be stored on any mass storage (directory) device and executed whenever desired. The loader also produces an optional symbol map that indicates the core storage requirements of the linked and relocated program. The overlay feature of the loader permits certain segments of a program to be stored in the loader image file during execution and read into core memory only as needed, which effectively provides a tenfold increase in maximum program size.

The loader image file produced by the loader is read and executed by system program FRTS.SV, the OS/8 FORTRAN IV run-time system, which also configures an I/O supervisor to handle any FORTRAN input or output in accordance with run-time I/O specifications. This makes the full I/O device independence of the OS/8 operating system available to every FORTRAN IV program, and permits FORTRAN programs to be written without concern for, or even knowledge of, the hardware configuration on which they will be executed. The run-time system assigns I/O device handlers to the I/O unit numbers referenced by the FORTRAN program, allocates I/O buffer space, and also diagnoses certain types of errors that occur when the loader image file is read into core. If no errors of this sort are encountered, the run-time system starts the FORTRAN program and monitors execution to check for run-time errors involving data I/O, numeric overflow, hardware malfunctions, and the like. Run-time errors are identified at the console terminal, and, when a run-time error occurs, the system also provides complete error traceback to identify the full sequence of FORTRAN statements that terminated in the error condition.

The compiler, assembler, loader, and run-time system each accept standard OS/8 Command Decoder option specifications, as do most OS/8 programs. The option specifications are alphanumeric characters which may be thought of as switches that, by their presence or absence, enable or disable certain program features and conventions. For example, specifying the /N option to the compiler suppresses compilation of internal sequence numbers, thereby reducing program memory requirements (at the cost of preventing full error traceback during execution). Thus, /N is one of the compiler run-time option specifications that may be requested to modify the usual compilation procedure. In this context, run time refers to the time at which the compiler, or other system program, is executed, rather than the time at which the FORTRAN program is executed.

A FORTRAN source program may be executed by first calling the compiler to convert the source into RALF assembly language, next calling the assembler to produce a relocatable binary file, then calling the loader to link and relocate the binary file, and finally calling the run-time system to load the program and supervise execution. OS/8 FORTRAN IV provides a program chaining feature that can simplify or eliminate this sequence of program calls in most cases. When chaining is requested, the first system program to be executed automatically calls the next program in the compiler/assembler/loader/run-time system sequence. When the compiler chains to the assembler, for example, the five programs (the compiler consists of four programs) function as a single unit that accepts FORTRAN source language input and generates relocatable binary output suitable for use as input to the loader. In this manner, simple FORTRAN programs may be compiled, assembled, relocated, loaded, and executed -- all as the result of a single Keyboard Monitor or CCL command. More complicated programs involving subroutines and, perhaps, overlays, do not admit to a high degree of chaining because a great deal of user input in the form of run-time option specifications may be required at some point in the chain. In general, however, it is usually most convenient to chain from the compiler to the assembler (combining compilation and assembly into a single operation) and from the loader to the run-time system (combining relocation, loading, and execution).

Errors encountered by the various system programs do not result in termination of program chaining unless the error is such that it is impossible for execution to continue. This permits the system to locate and identify as many errors as possible before returning control to the Keyboard Monitor. When chaining is requested, intermediate output files produced by one system program are deleted automatically after they have been read as input by the next program in the chain sequence. This serves to optimize storage requirements and minimize access time, particularly on DECtape- and LINCtape-based systems.

The OS/8 FORTRAN IV system also includes FORLIB.RL, a library of FORTRAN functions and subroutines, plus LIBRA, the system librarian program. Almost every FORTRAN program executes calls to library functions and subroutines which perform such tasks as mathematical function evaluation, data I/O, and numeric conversion. When the loader recognizes that a program or subroutine has called a library component, it copies a relocated version of the referenced library routine into the loader image file and links it to the calling routine. LIBRA is used to maintain the library by inserting or deleting library functions or subroutines, which are simply assembled FORTRAN files or specially coded RALF modules. LIBRA may also be used to create alternate libraries for use in place of the standard system library.

Because it affords full I/O device independence, highly optimized memory and bulk storage, program chaining, and a variety of run-time options, OS/8 FORTRAN IV is necessarily somewhat complicated. In order to use the system most efficiently, it is important to identify the four processes that must be performed, and their proper sequence, to execute a FORTRAN source program:

Process	Performed by
COMPILATION	FORTRAN IV compiler (F4, PASS2, PASS20 and PASS3).
ASSEMBLY	RALF assembler (RALF).
RELOCATION	FORTRAN loader (LOAD) using system library.
EXECUTION	FORTRAN run-time system (FRTS).

It is also important to identify the types of input that must be supplied to each process listed above and the types of output that will be produced. The OS/8 FORTRAN IV system accepts user-generated FORTRAN source programs (supplied as input to the compiler) and user-written RALF assembly language files (supplied to the assembler) as input. It generates four types of output files:

- RALF assembly language files generated by the compiler and read as input by the assembler. Compiler output is functionally equivalent to user-written RALF language input.
- Relocatable binary files generated by the assembler and read as input by the loader.
- Loader image files generated by the loader and read as input by the run-time system. Once a program has been written and debugged, it may be stored as a loader image file and executed whenever required without the necessity for further compilation, assembly, or relocation.
- Optional listing files including the FORTRAN source listing produced by the compiler, the RALF language listing produced by the assembler, and a symbol map produced by the loader.

In addition, the FORTRAN program itself usually reads and writes data files under the supervision of the run-time system; FORTRAN I/O files are treated separately in the section on the FORTRAN IV Run-Time System.

Every FORTRAN source program thus generates up to three object files, aside from any I/O files that may be read or written during execution, and up to three listing files. System-generated files are most conveniently identified by assigning them the same file name as the source from which they were produced and a file extension that identifies them by type. Table 1-1 lists the standard file extensions used to identify various types of source and system-generated files. The standard extensions are called default extensions because, when any output file name is specified with a null extension, the appropriate standard extension is appended by default. Thus, specifying file "PROG" or "PROG." to the RALF assembler, for example, causes the relocatable binary output from the assembly to be written on file "SYS:PROG.RL" where "SYS:" is the default device when a file name is explicitly defined and ".RL" is the default extension for relocatable binary files. Specifying a null file causes this output to be routed to file "DSK: FORTRN.RL" where "DSK:" is the OS/8 default device and "FORTRN" is the default output file name. For clarity, all examples in this chapter will use either null or default extensions, although the user may explicitly specify any extension desired.

Table 1-1
Standard FORTRAN IV File Extensions

Extension	File Type
.FT	FORTRAN language source file.
.RA	RALF assembly language file.
.RL	Relocatable binary (assembler output).
.LD	Loader image.
.LS	Listing or symbol map.
.TM	System temporary file. Created by certain multipass programs and normally deleted automatically after use.

This chapter assumes that the reader is familiar with the OS/8 operating system; however, all material has been presented in a manner that requires minimal experience with OS/8. The reader should understand the use of the OS/8 Keyboard Monitor (although only the monitor R command is referenced here) and the OS/8 Command Decoder. In particular, notice that all Command Decoder file/option specifications presented here are illustrated in a standard format that may not be the most convenient format for an experienced user's particular application. In addition, the Command Decoder provides file storage optimization features, which may be invaluable in many applications, but which are not covered in this chapter. DECtape and LINCtape users will benefit from an understanding of the OS/8 file structure, so that they may assign I/O files in a manner that minimizes access time on tape-based systems.

The FORTRAN IV system of programs may be entered through the CCL commands COMPILE, EXECUTE, and LOAD. These commands are described in Sections 1.1 and 1.1.1 in this chapter.

1.1 THE FORTRAN IV COMPILER

The OS/8 FORTRAN IV compiler accepts one FORTRAN source language program or subroutine as input, examines each FORTRAN statement for validity, and produces as output a list of error diagnostics, a RALF assembly language version of the source program, and an optional annotated source listing. A job containing one or more subroutines is run by compiling and assembling the main program and each subroutine separately, then combining them with the loader. F4 terminates a compilation by chaining to the RALF assembler automatically, unless it was requested to return to the Keyboard Monitor. The compiler is called by typing

```
.R F4
```

(terminated by a carriage return) in response to the dot generated by the Keyboard Monitor. F4 may also be called via the CCL command COMPILE. The compiler replies by loading the OS/8 Command Decoder, which accepts and decodes a standard command line that designates 0 to 3 output files, 1 to 9 input files, and any run-time option specifications. The file/option specification command line is entered by typing

```
DEV:RALF.RA,DEV:LIST.LS,DEV:MAP.LS<DEV:IF1.FT,...,DEV:IF9.FT(options)
```

(terminated by a carriage return or altmode) in response to the asterisk generated by the Command Decoder, where DEV:RALF.RA,DEV:LIST.LS, and DEV:MAP.LS are output files, RALF assembly source file, listing file, and loader symbol map file, respectively. The files DEV:IF1.FT,...,DEV:IF9.FT are input files 1 to 9. Options is a string of alphabetic characters, enclosed in parentheses, that designates any run-time options desired. The " " character may be used in place of the "<" character to separate output file specifications from input file specifications. The parentheses may be omitted if each run-time option specification character is preceded by a "/" character.

When any input file name is entered with a null extension, the compiler will search for the indicated file name with an assumed extension of ".FT". If this is unsuccessful, it will then search for the indicated file with a null extension. If the first output file RALF.RA is entered with a null extension, the compiler appends the default extension ".RA". If the second output file is a directory

device file with a null extension, the compiler appends the default extension ".LS". Note that unless chaining to RALF, the first output file is always written onto the OS/8 system device; any user device specification entered for this file will be ignored when the /A option is specified. When there is more than one input file, all of the input files are assumed to contain a single FORTRAN program or subroutine.

After accepting and decoding the file/option specification command, the compiler reads the input files in the order they were entered and then compiles each FORTRAN source statement until an END statement is encountered. Any text following the first END statement is ignored. The compiler then writes a RALF assembly language version of the source program onto the first output file, or onto file SYS:FORTRN.RA if no first output file was specified. It also copies an annotated source program listing onto the second output file; however, this listing is not produced unless a second output file was specifically defined. The third output file is not used by the compiler; it receives a loader symbol map only when chaining to the loader.

An internal statement number (ISN) is assigned to each FORTRAN IV statement sequentially, in octal, beginning with ISN 2 at the first FORTRAN statement. When an error is encountered during compilation, the compiler prints a 2-character error code, followed by the ISN of the offending statement, on the console terminal during pass 2. An extended error message is printed below every erroneous statement in the listing, provided that a listing is produced. Certain errors cause an immediate return to the Keyboard Monitor, however, in which case the listing file is never produced. Table 1-3 lists the FORTRAN compiler error messages and describes the error condition indicated by each message.

The compiler accepts five run-time option specifications, listed in Table 1-2, any combination of which may be requested by entering the appropriate alphabetic character(s) in the Command Decoder file/option specification line. Any run-time options recognized by the RALF assembler, the loader, or the run-time system may be entered along with the compiler options; they will be passed to the assembler automatically unless chaining is suppressed (by an error condition or the A option), in which case they will be ignored.

Table 1-2
FORTRAN IV Compiler Run-Time Options

Option	Operation
A	Return to the Keyboard Monitor when compilation is complete. If the A option is not requested, the compiler will automatically chain to the RALF assembler.
F	Produce an annotated listing of the RALF assembly language output file. The listing is actually produced by the assembler; thus, the F option is only valid when chaining to RALF. The listing is routed to the same output file as the FORTRAN source listing. It will overwrite the FORTRAN listing if the second output file resides on a directory device. It will not be produced if a second output file was not specifically defined.

(continued on next page)

Table 1-2 (Cont.)
FORTRAN IV Compiler Run-Time Options

Option	Operation
N	Suppress compilation of ISNs. This reduces program memory requirements by two words per executable statement; however it also prevents full error traceback at run time.
Q	Optimize cross-statement subscripting during compilation. This option should not be requested when any variable that appears in a subscript is modified either by referencing a variable equivalent to it or via a SUBROUTINE or FUNCTION call (whether as an argument or through COMMON).

1.1.1 Compiler Examples

Compile, assemble, load, and execute a FORTRAN IV source program:

```
.R F4
*PROG/G
```

Compiles DSK:PROG.FT or DSK:PROG into DSK:FORTRN.RA, assembles it into DSK:FORTRN.RL, links it into DSK:FORTRN.LD, then loads it into core and executes it. No listing files are produced.

Compile any source program by calling F4 and specifying the file (or files) containing the source as input:

```
.R F4
*PROG/A
```

Compiles DSK:PROG.FT or else DSK:PROG. into SYS:FORTRN.RA. The back-arrow is optional when there are no output file specifications.

```
.R F4
*SYS:PROG.FT(NA)
```

Compiles SYS:PROG.FT into SYS:FORTRN.RA under the N option.

Obtain a source listing with error messages by specifying a listing output file as the second output file. In these examples, the first output file is a null file.

```
.R F4
*,LPT:<PROG/A
```

Identical to the first of the two preceding examples, except that a listing is produced on the line printer.

```
.R F4
*,DTA1:PROG<DTA2:PROG.FT/A/N
```

Compiles DTA2:PROG.FT into SYS:FORTRN.RA and writes a source listing onto file DTA1:PROG.LS under the N option.

Designate a specific output file to receive the compiler output by specifying it as the first output file:

```
.R F4
*PROG<PROG/A
```

Compiles DSK:PROG.FT or else DSK:PROG. into SYS:PROG.RA.

```
.R F4
*WHEN.RA,WHERE.LS<RXA0:WHAT(AQ)
```

Compiles RXA0:WHAT.FT or else RXA0:WHAT. into SYS:WHEN.RA with a listing routed to DSK:WHERE.LS under the Q option.

1.1.2 Compiler Error Messages

During compilation pass 2, error messages are printed at the console terminal as a 2-character error message followed by the ISN of the erroneous statement. Typing CTRL/O at the terminal suppresses the printing of error messages. During optional pass 3, which requests a listing, an extended error message follows each erroneous statement on the listing. Except where indicated in Table 1-3, errors located by the compiler do not halt processing.

Table 1-3
FORTRAN IV Compiler Error Messages

Error Code	Meaning
AA	More than six subroutine arguments are arrays.
AS	Bad ASSIGN statement.
BD	Bad dimensions (too big, or syntax) in DIMENSION, COMMON, or type declaration.
BS	Illegal in BLOCK DATA program.
CL	Bad COMPLEX literal.
CO	Syntax error in COMMON statement.
DA	Bad syntax in DATA statement.
DE	Illegal statement as end of DO loop (i.e., GO TO, another DO).
DF	Bad DEFINE FILE statement.
DH	Hollerith field error in DATA statement.
DL	Data list and variable list are not same length.
DN	DO-end missing or incorrectly nested. This message is not printed during pass 3. It is followed by the statement number of the erroneous statement, rather than the ISN.
DO	Syntax error in DO or implied DO.
DP	DO loop parameter not integer or real.
EX	Syntax error in EXTERNAL statement.
GT	Syntax error in GO TO statement.
GV	Assigned or computed GO TO variable not integer or real.
HO	Hollerith field error.
IE	Error reading input file. (Control returns to the Keyboard Monitor.)
IF	Logical IF statement used with DO, DATA, INTEGER, etc.
LI	Argument of logical IF not type Logical.
LT	Input line too long, too many continuations.
MK	Misspelled keyword.
ML	Multiply defined line number.
MM	Mismatched parenthesis.
MO	Expected operand is missing.
MT	Mixed variable types (other than integer and real).
OF	Error writing output file. (Control returns to the Keyboard Monitor.)
OP	Illegal operator.
OT	Type / operator use illegal (e.g., A.AND.B where A and / or B not typed Logical).
PD	Compiler stack overflow; statement too big and/or too many nested loops.
PH	Bad program header line.
QL	Nesting error in EQUIVALENCE statement.
QS	Syntax error in EQUIVALENCE statement.
RD	Attempt to redefine the dimensions of a variable.

(continued on next page)

Table 1-3 (Cont.)
FORTRAN IV Compiler Error Messages

Error Code	Meaning
RT	Attempt to redefine the type of variable.
RW	Syntax error in READ/WRITE statement.
SF	Bad arithmetic statement function.
SN	Illegal subroutine name in CALL.
SS	Error in subscript expression, i.e., wrong number, syntax.
ST	Compiler symbol table full, program too big. (Causes an immediate return to the Keyboard Monitor.)
SY	System error, i.e., PASS20.SV or PASS2.SV missing, or no room on system for output file. (Causes an immediate return to the Keyboard Monitor.)
TD	Bad syntax in type declaration statement.
US	Undefined statement number. This message is not printed during pass 3. It is followed by the statement number of the erroneous statement, rather than the ISN.
VE	Version error. One of the compiler programs is absent from SYS: or is present in the wrong version.

1.2 THE RALF ASSEMBLER

The RALF assembler accepts one RALF assembly language program or subroutine as input and produces a relocatable binary file, called a RALF module, as output. An optional annotated listing of the input file may also be produced. RALF terminates an assembly by returning to the Keyboard Monitor unless it was requested to chain to the loader.

A RALF module is composed of an external symbol dictionary (ESD table) and associated text. The ESD table lists all symbols defined in the RALF input file, which may be sections, entry points, or externs. Each of these symbols is assigned a relative address to be used by the loader when it relocates the relative code by assigning absolute core addresses. The text produced by RALF is a relocatable binary version of the assembly language input file. All text addresses are relative to the ESD table symbols.

A section can be thought of as a contiguous block of relocatable code having a definite beginning and end, which is temporarily assigned a relative starting address of 00000. A RALF file can have more than one section defined in its ESD table. For example, consider a subroutine containing a COMMON section assembled by RALF. Both COMMON and the subroutine itself are sections. An entry point is a location within a given section that is referenced by code in other sections. An extern is a section or entry point in some other module that is referenced within the module currently being assembled.

Unless the A option is specified to the FORTRAN IV compiler, the RALF assembler is called automatically to assemble the output of a successful compilation. In this case, RALF reads the assembly language file just produced by the compiler as input and routes its output, consisting of the assembled RALF module, to the first output file that was specified to the compiler. If this file had a null extension, the default extension ".RL" is supplied. If no first output file was specified, the module is written onto default file SYS:FORTRN.RL.

The RALF language output produced by the compiler is then deleted, and an annotated listing of the RALF assembly language input is written on the second output file specified to the compiler, provided that a second output file and the F option were both specified. This listing will overwrite the compiler source listing if the second output file is a directory device file. Note, however, that the RALF language listing is rarely required for most applications and should not be routinely requested.

The RALF assembler might also be called separately to assemble the output of the compilation produced under the A option or to assemble a user-generated file written in RALF assembly language. This is accomplished by typing

```
.R RALF
```

(terminated by a carriage return) in response to the dot generated by the Keyboard Monitor. RALF replies by loading the OS/8 Command Decoder, which accepts and decodes a standard command line that designates 0 to 3 output files, 1 to 9 input files, and any run-time option specifications. The format for a file/option specification command line is

```
DEV:RALPH.RA,DEV:LIST.LS,DEV:MAP.LS<DEV:IF1.RA,...,DEV:IF9.RA(options)
```

where

DEV:RALF.RA — is the relocatable binary RALF module

DEV:LIST.LS — is the annotated listing of RALF source

DEV:MAP.LS — is the loader symbol map

DEV:IF1.RA,...,DEV:IF9.RA — are input files 1 to 9

options — is a string of alphabetic characters that designates any run-time options desired

If any input file name is entered with a null extension, the assembler will search for the indicated file name with an assumed extension of ".RA". If this is unsuccessful, it will then search for the indicated file with a null extension. If the first output file is entered with a null extension, the assembler appends the default extension ".RL". If the second output file is a directory device file with a null extension, the assembler appends the default extension ".LS".

When there is more than one input file, all of the input files are assumed to contain the assembly language source for a single RALF module. After accepting and decoding the file/option specification command, RALF reads the input files in the order they were entered and assembles every RALF language statement. RALF terminates the assembly by writing a relocatable binary version of the input program or subroutine onto the first output file, or onto file SYS:FORTRN.RL if no output files were specified. It also copies an annotated source listing and symbol table onto the second output file; however, this listing is not produced unless a second output file was specifically defined. The third output file is not used by the assembler; it receives a loader symbol map only when chaining to the loader.

When an error is encountered during assembly, the assembler prints a 2-character error code, followed by the label associated with the erroneous statement, on the console terminal during pass 2. Error codes are also appended to the listing, on a line by themselves

immediately preceding the statement to which they apply (except EG, which follows the line in error). Certain errors cause an immediate return to the Keyboard Monitor, however, in which case the listing is never produced. RALF assembler error messages and the error condition indicated by each message are described in the RALF chapter of this manual.

The assembler accepts the three run-time option specifications listed in Table 1-4, any combination of which may be requested by entering the appropriate alphabetic character(s) in the Command Decoder file/option specification line. Any options recognized by the loader or the run-time system may be entered along with the assembler options; they will be passed to the loader automatically unless chaining is suppressed (by an error condition or omission of the L option specification), in which case they will be ignored.

Table 1-4
RALF Assembler Run-Time Options

Option	Operation
G	Chain to the loader when assembly is complete, and chain to the run-time system following creation of a loader image file.
L	Chain to the loader when assembly is complete. If the L option is not specified, RALF will return to the Keyboard Monitor upon completion.
T	Suppress the RALF assembly language listing and produce only a symbol table. The T option is ignored by the assembler when a second input file was not specifically defined. When chaining from the compiler, it is ignored unless the F option and a listing output file were both specified.

The symbol table produced by RALF and appended to the RALF language listing includes:

- assembler version number
- system date
- listing page number
- number of errors encountered during assembly
- number of symbols defined in the program
- number of absolute references encountered in FPP instructions

All symbols referenced during the assembly are then listed in alphabetical order, from left to right across the page. An alphabetic code follows certain classes of symbols and identifies them by type. The alphabetic codes are:

C = symbol names a COMMON section
F = symbol names a FIELD1 section
S = symbol is the name of a section
U = symbol is undefined
X = symbol is external to this assembly
Z = symbol names a COMMZ section
8 = symbol names an 8-mode section

If no alphabetic code is shown, the symbol is an ordinary address symbol. A numeric code is also printed after each symbol in the list. The numeric code indicates the relative octal value of the symbol except for the case of:

C, F, S, Z, or 8 codes	where the numeric code indicates the length of the section or common block.
U or X codes	where 00000 indicates undefined or external symbols.

1.2.1 RALF Examples

When chaining from the compiler to the assembler, RALF deletes the compiler output after reading it as input. Thus:

.R F4 *PROG	Produces RALF module SYS:FORTRN.RL and deletes compiler output file SYS:FORTRAN.RA.
.R F4 *PROG.V3,LPT:<PROG/F	Produces RALF module SYS:PROG.V3 and lists both the FORTRAN source and the RALF language compiler output on the line printer.
.R F4 *DATA2:OBJ,DTA1:LIST<DTA2:PROG(TF)	Produces RALF module DTA2:OBJ.RL and writes a symbol map onto file DTA1:LIST.LS. The FORTRAN source listing is overwritten and destroyed.

When calling the assembler to assemble and relocate the output of a successful compilation produced under the A option or a user-written RALF language source, the procedure is closely analogous to that for running the compiler:

.R RALF *PROG	Assembles DSK:PROG.RA or else DSK:PROG. into SYS:FORTRN.RL.
.R RALF *,SYS:LIST<DTA1:FILE.RA	Assembles DTA1:FILE.RA into SYS:FORTRN.RL and writes a listing on SYS:LIST.LS.
.R RALF *DTA1:TMP.TM,LPT:<RALF.RA	Assembles DSK:RALF.RA into DTA1:TEMP.TM and writes a listing on the line printer.

1.2.2 RALF Assembler Error Messages

During assembly pass 2, error messages are printed at the console terminal as a 2-character error code followed by the label associated with the erroneous statement. If a listing was requested, error codes are printed during pass 3 on a line by themselves immediately preceding the statement to which they apply (except for EQ, which follows the line in error). RALF error messages are listed in the RALF chapter of this manual.

1.3 THE LOADER

The OS/8 FORTRAN IV loader accepts up to 128 RALF modules as input and links the modules, along with any necessary library components, to form a loader image file that may be loaded and executed by the run-time system. This is accomplished by replacing the relative starting location (00000) of each section with an absolute core address. Absolute addresses are also assigned to all entry points defined in the input modules. Once all RALF modules and library components have been assigned to some portion of memory and linked, absolute addresses are assigned to the relocatable binary text and the externs.

The overlay feature of the loader facilitates running programs too large to be contained in available memory. This makes it possible to run programs that require up to 300K words of storage in less than 32K of actual core memory. This is accomplished by dividing very large FORTRAN programs into a set of subroutines linked by one mainline. Unlike the subroutines, each of which has a section name by which it is called, the mainline does not have a name and is therefore assigned section name #MAIN by the system. An overlay scheme is then designed in such a way that the memory requirement of those subroutines that are core-resident at any given time does not exceed the available core memory.

An overlay is a set of subroutine stored on a bulk storage device. When any subroutine in an overlay is called by the mainline or another subroutine, the entire overlay is read into core, where it generally replaces another overlay of equivalent size.

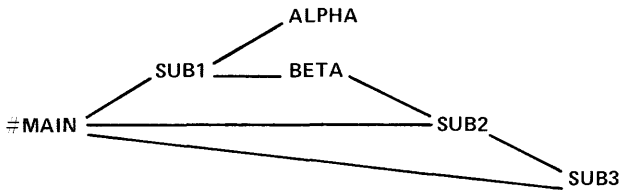
Levels are variable-size portions of memory reserved for specific sets of overlays. OS/8 FORTRAN IV permits up to 8 levels, designated level 0, level 1, and so on up to level 7. Level 0 is always present and always contains only one overlay, called overlay MAIN, which always includes section #MAIN (the FORTRAN or RALF mainline) as well as all COMMON sections, 8-mode sections and library components. Additional subroutines may also reside in overlay MAIN; in fact, the entire program should be loaded into level 0 if there is sufficient core available.

Levels 1 to 7 may each contain up to 16 overlays, only one of which is core-resident at any given time during program execution. If no subroutines are loaded into a given level, that level does not exist for the current execution and no memory is allocated to it. As execution begins, overlay MAIN is loaded into level 0 (where it remains throughout execution) and started at the entry point of section #MAIN. Other overlays are read into the block of memory reserved for their particular level whenever one of their constituent subroutines is called. As an overlay is read into a given level, it overwrites any other overlay that may have been resident in that level. Thus, no two overlays from the same level are ever core-resident simultaneously.

When section #MAIN or any subroutine calls another subroutine, the flow of execution from calling routine to called routine is referred to as part of a calling sequence. Every calling sequence begins with a call from section #MAIN and ends with a call to some subroutine that does not contain any further CALL statements. Calling sequences generally contain branches, and they may be very intricate. For example, assume that:

Routine/Subroutine	Contains Calls To
mainline (#MAIN)	SUB1, SUB2, SUB3
SUB1	ALPHA, BETA
SUB2	SUB3
SUB3	
ALPHA	
BETA	SUB2

Then the calling sequences could be mapped as:



When any subroutine CALL is executed, the system determines whether the overlay containing the called routine is core-resident and, if not, reads this overlay into its proper level in core, overwriting any overlay which was previously resident in that level. No such determination is possible for RETURN statements, however. For this reason, it is extremely important to ensure that, at the end of a calling sequence, all subroutines in the calling sequence are still core-resident. In other words, no subroutine may execute a CALL that will cause it, or any subroutine which called it, to be overlaid. In the previous example, if SUB1, SUB2 and SUB3 occupy separate overlays in level 1 while ALPHA and BETA reside in level 2, the calling sequence from #MAIN to SUB1 to BETA to SUB2 will cause a fatal error because SUB2 will overwrite SUB1 and prevent control from returning to level 0. The FORTRAN system guards against some errors of this type by enforcing the following rules:

- Subroutines in a given level cannot call other subroutines in the same level if the called subroutine is in a different overlay.

- Subroutines in high-numbered levels cannot call subroutines in lower-numbered levels unless the call is to level 0. (This convention is not enforced when the U option is specified to the run-time system.)

These restrictions will not prevent fatal errors in all cases. In the preceding example, if subroutine BETA is placed in level 0 instead of level 1, the calling sequence from #MAIN to SUB1 to BETA to SUB2 still causes a fatal error, even though neither of the enforced conventions is violated. Thus, any overlay scheme must be designed with careful attention to calling sequences.

If the L or G option is specified to F4 or RALF, the loader is called automatically to relocate the output of a successful assembly. When chaining to the loader is via F4, the loader reacts in one of two ways. If the last Command Decoder file/option line terminated with a carriage return, it immediately fetches the Command Decoder and proceeds as though it had been called from the monitor, as described below. The only difference, in this case, is that certain loader or run-time system options may have been passed to the loader from RALF and cannot be suppressed at this point. Also, unless two different files are specified as output files, the loader automatically routes its loader image to the first output file specified to F4 or RALF at the start of the chain. Default extension ".LD" is assigned if this file had a null extension. If no output files were specified the loader routes its loader image to file SYS:FORTRAN.LD. The relocatable binary output produced by the assembler is deleted after it has been read as input. A loader symbol map is routed to the third output file specified at the start of the chain sequence, if any, or to the second output file, if any, specified to the loader as described below. When this is a directory device file with a null extension, the default extension ".LS" is supplied.

If the last file/option specification supplied to the Command Decoder was terminated with an ALTMODE character instead of a carriage return, the loader reacts differently when chained to from RALF. In this case, the loader assumes that the RALF module just produced is a stand-alone mainline that requires no subroutines (other than library components) in order to execute. The loader does not call the Command Decoder under these circumstances, since level 0 is the only level that will be defined. Output is produced exactly as described above, and the loader either returns to the Keyboard Monitor upon completion or, if a G option specification was previously entered, chains to the run-time system.

The loader may be called separately, to link and relocate a set of previously assembled RALF modules. This is accomplished by typing

```
.R LOAD
```

(terminated by a carriage return) in response to the dot generated by the Keyboard Monitor. The loader replies by calling the OS/8 Command Decoder, which accepts and decodes one or more standard command lines, each of which designates 0 to 9 input files, 0 to 2 output files, and any run-time option specifications desired. The file/option specification line format is:

```
DEV:IMAGE.LD,DEV:MAP.LS<DEV:PROGA.RL,...,DEV:PROGX.RL(options)
```

where

IMAGE.LD is the loader image output file

MAP.LS is the loader symbol map output file

DEV:PROGA.RL,...,DEV:PROGX.RL
may be either relocatable binary RALF modules or a library file

options is a string of alphabetic characters that designates any run-time options desired

The loader accepts up to 128 input file specifications, one of which may designate a library file to be used in place of the standard system library. The OS/8 Command Decoder, however, accepts a maximum of only 9 input file specifications per command line. Thus, after

each file/option command line is entered, the loader recalls the Command Decoder to accept another command line. This process continues until the /G option is received or a line is terminated with an ALTMODE. Input file specifications should be entered in sequence, beginning with all RALF files to be loaded into level 0, followed by files for level 1 overlay 1, level 1 overlay 2, and so on until all level 1 overlays are filled. Level 2 overlays are then built in the same manner, using as many file/option specification lines as necessary. The process continues until all levels are filled. Each line may contain from 0 to 9 input file specifications; null lines will be ignored by the loader.

At some point during this process, two output files and one library (input) file may also be specified. The loader image file built by the loader is routed to the first output file, which must reside on a directory device, or to file SYS:FORTRN.LD if no output files are specified. When the first output file has a null extension, the default extension ".LD" is supplied. The loader symbol map is routed to the second output file, provided that a second file is specifically defined. If this is a directory device file with a null extension, the default extension ".LS" is supplied. One library file may be specified as an input file, to be used in place of the standard system library. This must be a specially formatted file, prepared with LIBRA as described in Chapter 13 of this manual. In addition, it must be specified on a command line that contains no other input file names. This command line may appear anywhere in the file/option specification sequence and is identified by the presence of an L option specification.

If more than one first output file, second output file, or library file is specified to the loader, only the last specification in each category is used. Previous specifications, including those supplied to F4 or RALF when chaining to the loader, are ignored.

Run-time option specifications are used to group the sequence of input files into discrete overlays, allocate overlays to certain levels, and identify the user-generated library file, if any. Table 1-5 lists the run-time options recognized by the loader and describes their use. The E and H options, recognized by the run-time system, may be entered on the same line as the G option when chaining to the run-time system.

Table 1-5
Loader Run-Time Options

Option	Operation
C	Continue the current line of input on the next line of input. When specifying RALF files to the loader, there may be more than nine files that belong in a given overlay. Since the Command Decoder will not allow more than nine input files in one file/option specification line, the C option permits the additional files to be put on the following line. If the C option is not specified at the end of a line, the current overlay is closed when the terminating carriage return is received and subsequent input files are placed in a new overlay in the current level. An exception to this is level 0, which only contains one overlay. The presence of a C option specification is assumed on every line until level 0 has been closed by an 0 specification.

(continued on next page)

Table 1-5 (Cont.)
Loader Run-Time Options

Option	Operation
G	Treat the current line as the last line of input, and chain to the FORTRAN IV run-time system when finished.
L	Accept the single input file specified on this line as an alternate library to be used in place of the system library, FORLIB.RL.
O	Close the level that is currently open, and open the next sequential level for input. RALF files specified on subsequent lines are assigned to overlays in the new level until the new level is closed by the next O specification (or the end of input).
S	Include system symbols in the loader symbol map. System symbols are identified by an initial "#" character. This option is only valid when a symbol map output file was specifically defined.
U	Ignore the rules governing subroutine calls between overlays. This option should only be used when subroutines making illegal calls will not be accessed during execution since, in general, any illegal subroutine call will cause unpredictable behavior at run time.

Input may be terminated by entering a G option specification on the last line and/or by terminating the last line with an ALTMODE character rather than a carriage return. If the G specification and the ALTMODE both appear, this indicates that the user has no file/option specification input for the run-time system and prevents the run-time system from calling the Command Decoder.

1.3.1 Loader Examples

The following sequence of Command Decoder specification lines illustrates the use of option specifications to allocate RALF files to particular overlays.

```
.R LOAD
*SYS:PROG.LD,LPT:<PROG.RL
*<ALPHA.RL,BETA.RL
```

Loader is called from Keyboard Monitor.

Loader image file will be routed to SYS:PROG.LD while the symbol map is printed on the line printer. PROG.RL is placed in level 0 overlay MAIN. Since the presence of a C option specification is assumed on every line preceding the first O option specification, level 0 overlay MAIN remains open.

Place subroutines ALPHA and BETA in level 0 overlay MAIN. The presence of a C option specification is assumed.

*^/O	Close level 0 and open level 1 overlay 1.
*^<SUB1.RL,SUB2.RL,SUB3.RL	Place SUB1, SUB2 and SUB3 in level 1 overlay 1. Close overlay 1 and open overlay 2.
*^<SUB4.RL,SUB5.RL,SUB6.RL/C	Place SUB4, SUB5 and SUB6 in level 1 overlay 2. Accept further input for this overlay on the next line.
*^<DTA1:SUB7.RL/O	Place SUB7 in level 1 overlay 2. Close level 1 and open level 2 overlay 1.
*^<DTA1:SUB8.RL	Place SUB8 in level 2 overlay 1. Close overlay 1 and open overlay 2.
*^<SUB9.RL	Place SUB9 in level 2 overlay 2. Close overlay 2 and open overlay 3.
*^<LIB.RL(LS)	Use file DSK:LIB.RL in place of SYS:FORLIB.RL as the library file. In spite of its position in the specification list, any library components will be placed in level 0. The S option specification requests an augmented loader symbol map.
*^<SUB10.RL/O	Place SUB10 in level 2 overlay 3. Close level 2 and open level 3 overlay 1.
*^<SUB11.RL,DTA1:SUB12.RL/G	Place SUB11 and SUB12 in level 3 overlay 1. Close level 3, terminate input, and chain to the run-time system when finished.

This sequence of commands will provide the following overlay scheme:

Level	Overlay	Contents
0	MAIN	PROG, ALPHA, BETA library subroutines
1	1	SUB1, SUB2, SUB3
1	2	SUB4, SUB5, SUB6, SUB7
2	1	SUB8
2	2	SUB9
2	3	SUB10
3	1	SUB11, SUB12

Note that all of the input files except those containing SUB7, SUB8, and SUB12 are taken from device DSK:, the OS/8 default device. The left-angle bracket character is optional when a file/option specification line contains only input file specifications; it has been included here for clarity. Obviously, there are many other ways in which the sequence of file/option specifications shown above could have been entered to produce an identical result.

Considerable foresight is required when designing an overlay scheme. Since an overlay may have to be read into core whenever one of its constituent subroutines is called, a great deal of useless I/O results from inefficient overlay design. The system does verify that an overlay is not already resident before reading it into core.

Levels must be an integral number of system blocks (400 octal words in size) and big enough to accommodate the largest overlay they contain.

Ideally, then, the largest overlay in a level should occupy slightly less than some multiple of 400 (octal) words of storage, and all overlays in a level should be nearly equal in size. For example, if level 1 contains three overlays requiring 300, 100, and 150 octal words of storage, respectively, then the two smaller overlays should be combined because level 1 will be 400 octal words long in any case. If the three overlays require 500, 100, and 150 octal words of storage, all three should be combined because level 1 will be 1000 octal words long in any case.

Frequently called subroutines should be kept core-resident whenever possible, perhaps by placing them in level 0 or in a level that contains rarely accessed overlays. Within the loader image file, subroutines are stored in the order in which they were specified to the loader. Thus, grouping frequently called subroutines into adjacent levels also speeds execution by reducing the access time required to read an overlay into core, particularly from DECtape and LINCtape. When running very large programs with many overlay levels, it may be desirable to make level 0 as small as possible, in spite of the resulting excess I/O. This is accomplished by minimizing COMMON (which always occupies level 0), dividing the mainline into a series of subroutines, and creating a new mainline that contains predominately CALL statements. Note, however, that all library subroutines will reside in level 0, regardless of the location of subroutines that call them.

Any error recognized by the loader during generation of a loader image file results in an error message, printed on the console terminal, immediately following the input specification line that caused the error condition. Table 1-6 lists the loader error messages and describes the error condition indicated by each message.

The optional loader symbol map lists all symbols defined in the loader image file and identifies each symbol by overlay, level, and memory address, as follows:

```
LOADER V21  04 /30 /73

SYMBOL VALUE LVL OVLY

A      10400  1  00
ARGERR 00204  0  00
B      10400  1  01
C      11214  1  01
EXIT   00223  0  00
#MAIN  10000  0  00
 12000  = 1ST FREE LOCATION


LVL OVLY LENGTH

0  00   10143
1  00   01270
1  01   01240
```

Following the alphabetical list of symbols, the loader prints the address of the first free memory location and the length, in octal words, of each overlay defined. This information is useful in optimizing memory requirements.

1.3.2 Loader Error Messages

The loader prints error messages on the console terminal during generation of a loader image file. Except where indicated in Table 1-6, loader errors are fatal. The loader returns control to the Keyboard Monitor when a fatal error condition is encountered.

Table 1-6
Loader Error Messages

Error Message	Meaning
BAD INPUT FILE	An input file was not a RALF module.
BAD OUTPUT DEVICE	The loader image file device was not a directory device, or the symbol map file device was a read-only device. The entire line is ignored.
ILLEGAL ORIGIN	A RALF routine tried to store data outside the bounds of its overlay.
MIXED INPUT	The L option was specified on a line that contained some file other than a library file. The library file (if any) is accepted. Any other input file specification is ignored.
MULT SECT	Any combination of entry point, COMMON section (with the exception of multiple COMMONs), or program section of the same name causes this error, except the following: SECT: OK / OK / OK SECT8: OK / OK / OK COMMON: OK / (MS) / OK COMMZ: (MS) / OK / (MS) FIELD1: OK / (MS) / OK (COMMON / COMMZ / FIELD1)
NO MAIN	No RALF module contained section #MAIN.
OVER CORE	The loader image requires more than 32K of core memory.
OVER IMAG	Output file overflow in the loader image file.
OVER SYMB	Symbol table overflow. More than 253 (decimal) symbols in one FORTRAN job.
TOO MANY LEVELS	The O option was specified more than seven times.

(continued on next page)

Table 1-6 (Cont.)
Loader Error Messages

Error Message	Meaning
TOO MANY OVERLAYS	More than 16 overlays were defined in the current level.
TOO MANY RALF FILES	More than 128 input files were specified.
EX	The symbol is referenced but not defined.
ME	Multiple Entry. The symbol has more than one definition.
MS	Multiple Section. A section has more than one definition.
*	The symbol is referenced illegally. Generally this symbol is an overlay and is either referenced as data from another overlay (only CALL references are allowed) or called from the same or a higher-number overlay level, violating the overlay rules.

The following FATAL error messages occur when the Loader is linking and relocating:

SYSTEM ERROR

LOADER I/O ERROR

OS/8 ENTER ERROR

and indicate an error detected by OS/8 while trying to perform a USR function.

All errors identified during the loading procedure are followed by a line of the form:

l oo nnn

where

l is the level in which the error occurred

oo is the overlay in which the error occurred

nnn is the module number, within the referenced overlay, that caused the error.

Some errors (e.g., NO MAIN) are attributable to a single module, and the module numbers for this type of error are meaningless.

1.4 FORTRAN IV RUN-TIME SYSTEM (FRTS)

The OS/8 FORTRAN IV run-time system reads, loads, and executes a loader image file produced by the loader. It also configures a software I/O interface between the FORTRAN IV program and the OS/8

operating system, then monitors program execution to direct I/O processes and identify certain types of run-time errors. The run-time system is called automatically to load and execute the loader image file produced by the loader whenever the G option is specified to the loader.

When chained to from F4, RALF, or LOAD, the run-time system reacts in one of two ways. If the last Command Decoder file/option line was terminated with a carriage return, it immediately fetches the Command Decoder and proceeds as though it had been called from the Keyboard Monitor, as described below. The only difference, in this case, is that certain run-time system options may have been passed to the run-time system from the loader and cannot be suppressed at this point. If the last file/option specification line supplied to the Command Decoder was terminated with an ALTMODE character instead of a carriage return, however, the loader assumes that no user input is required. The Command Decoder is not called. The loader image file just produced is read as input, and, unless the H option was previously specified, it is loaded and executed.

The FORTRAN IV Run-Time System is able to accept file I/O specifications. This allows the user to write a source program that refers to I/O devices as integer constants or variables. This program may be compiled, assembled, and loaded into an image file. The image file may be run any number of times, each time specifying different physical I/O devices. Thus logical unit 8 may refer in one run to the console terminal, in another run to a disk file, and in another run to a paper tape punch.

These run-time specifications allow the FORTRAN program to use the OS/8 file-handling capabilities, to use any OS/8-supported I/O device, and potentially to use any I/O device for which an OS/8 device handler can be written.

The following pages explain how the user gives the run-time system the connections between OS/8 device and file names and the FORTRAN logical unit numbers.

FORTRAN IV programs are usually saved as loader image files and executed by calling the run-time system from the Keyboard Monitor to load and execute the saved loader image. This is accomplished by typing

```
.R FRTS
```

(terminated by a carriage return) in response to the dot generated by the Keyboard Monitor. The run-time system replies by calling the OS/8 Command Decoder to accept one or more standard file/option specification lines. It recalls the Command Decoder after processing each line, until a line terminated by an ALTMODE character is received.

The run-time system accepts two classes of Command Decoder file/option specifications. The first class specifies the load module to be executed; the second class specifies the run-time file assignment. When it is called from the Keyboard Monitor, the run-time system loads the Command Decoder to accept one input file name, perhaps followed by the E or H option specifications, described in Table 1-7. This information is not required when the loader chains to the run-time system because the loader image file just produced is automatically read as input, while the E and/or H options could have been specified to the loader along with the G specification that requested chaining.

Thus, the loader image input file to be executed must be identified on the first file/option specification line when FRTS is called from the Monitor, and must not be specified at all when the loader chains to FRTS. This Command Decoder line has the form:

*DEV:IMAGE.LD(options)

where IMAGE.LD is the loader image input file and "options" is E or H or both. If this line is terminated by an ALTMODE, the program is executed; if it is terminated with a carriage return, the Command Decoder is recalled to accept run-time file specifications.

Once the loader image file to be executed has been identified, the run-time system recalls the Command Decoder to accept any FORTRAN I/O device specifications. Of the nine I/O unit numbers available under FORTRAN IV, four are initially assigned to FORTRAN internal device handlers by the system as follows:

I/O Unit	Internal Handler	Comments
1	paper tape reader	Single-character buffer
2	paper tape punch	Single-character buffer
3	line printer	LP8 and LS8E only; ring buffered
4	console terminal	Double-buffered output; single-character input

The FORTRAN internal handlers listed above are not the same as the OS/8 device handlers. The FORTRAN internal handlers are designed for ASCII text only and will not execute binary or core-image I/O. Also, FORTRAN internal handlers are interrupt-driven to execute foreground I/O concurrently with background computation.

FORTRAN internal device handlers may be assigned different unit numbers, in addition to those listed above, by typing

/n=m

where

m is the I/O unit number (1 to 4) of one of the internal handlers listed above

n is a different unit number (1 to 9) that is also to be assigned to that internal handler

This specification causes all program references to logical unit n to perform I/O to device m in the preceding table. For example:

/6=2 Assigns the FORTRAN internal paper tape punch handler as I/O unit number 6, in addition to unit number 2.

/1=2 Assigns I/O unit number 1 to the FORTRAN internal paper tape punch handler instead of the internal paper tape reader handler.

OS/8 device handlers for nondirectory devices may be assigned I/O unit numbers by typing

DEV:/n

where

n is an I/O unit number (1 to 9)

DEV: is the standard or assigned designation for any supported nondirectory device

For example:

LPT:/3 Specifies the OS/8 line printer handler to be used instead of the FORTRAN internal line printer handler, possibly because the line printer is not an LP08 or LS8E.

Existing directory device files may be assigned I/O unit numbers by typing

DEV:FILE.EX/n

where

n is an I/O unit number (1 to 9)

DEV:FILE.EX is the standard OS/8 designation for an existing directory device file

For example:

*DTA1:FORIO.TM/2 Assigns unit number 2 to DECtape file FORIO.TM rather than to the FORTRAN internal paper tape punch handler, where FORIO.TM is an existing file on DECtape unit 1.

A directory device file that does not presently exist may be assigned a FORTRAN I/O unit number in the same manner by entering it as an output file on the specification line; however, only one such file may be created on any particular device. For example:

*FORIO.TM</9 Assigns unit number 9 to file DSK:FORIO.TM, which has not been created at load time.

In any case, only one device or file specification is permitted on each line, and no more than 6 directory device files may be created by the FORTRAN program. Excess files after the sixth are accepted and written, but they will not be closed. If a file created by the program has the same file name and extension as a pre-existing file, the old file is automatically deleted when the new file is closed.

The Command Decoder "[n]" specification may be used to optimize storage allocation when assigning files that do not yet exist, where n is a decimal number that indicates the maximum expected length of the file, in blocks.

Each time a run-time I/O specification is terminated with a carriage return, the Command Decoder is recalled to accept another specification. When a specification is terminated with an ALTMODE, the program is run.

Although existing files are specified as though they were input files and nonexistent files are specified as though they were output files, any file that has been assigned a unit number may be used for either input or output. The content of a nonexistent file is undefined until it has been written by the program.

Table 1-7
Run-Time System Option Specifications

Option	Operation
H	Halt after loading but before starting the program. Press the CONTinue switch on the processor to commence execution.
E	Ignore the following run-time system errors, any one of which indicates that an error was detected earlier in the compilation/assembly/loading process: a. Illegal subroutine call b. Reference to an extern in an overlay other than in the form "JSR EXTERN" (i.e., CALL statement) c. Reference to an undefined symbol Any of the above may lead to unpredictable program behavior as, in general, some portion of the program will not be loaded or executed.
C	Carriage control switch. The first character on every output line is processed as a carriage control character by all FORTRAN internal handlers and also by the OS/8 hard copy handlers TTY and LPT. The first character on every output line is processed as data, in the same manner as any other character, by all OS/8 handlers except TTY and LPT. Entering a C option specification on the command line that assigns an I/O unit number to a particular handler reverses the processing of carriage control characters for that device. Thus: TEMP(2C) assigns file DSK:TEMP. as I/O unit 2. The C option causes the first character of every output line to be processed as a carriage control character. If C were not specified, these characters would be processed as data. /C/6=3 assigns the FORTRAN internal line printer handler as I/O unit 6, as well as unit 3. The first character of every line will be processed as a carriage control character on unit 3, and as a character of data on unit 6.

The OS/8 FORTRAN IV run-time system executes with the PDP-8/E interrupt system enabled. OS/8 device handlers are not interrupt-driven; however, certain handlers may execute with the interrupt system enabled because the devices they control have interrupt-enable switches that the handlers do not set. FRTS allows

for this by running with the interrupt system enabled when driving handlers of this type, and disabling the interrupt system when a handler that does not run under interrupts is loaded. Handlers that can run with the interrupt system enabled include:

TC08 DECtape system handler and nonsystem handlers DTA0 to DTA7

RF08 system handler

RK8 system handler and nonsystem handlers RKA0 to RKA3

RK8E system handler and nonsystem handlers RKA0 to RKA3 and RKB0 to RKB3

Any FORTRAN internal handlers

These OS/8 handlers do not permit interrupts from these devices, but they do permit other devices, e.g., CLOCK, to interrupt the data transfer. Note that TD8E is absent from this list because the TD8E data transfer cannot be interrupted.

The run-time system recognizes two classes of error conditions. Certain errors are diagnosed while the core-image file is being read from a storage device and loaded into core memory. Other errors may occur during execution of the FORTRAN program. Both classes of run-time errors are identified on the console terminal. Table 1-8 lists the FRTS error messages and describes the error condition indicated by each message. The run-time system error traceback feature provides automatic printout of statement numbers corresponding to the sequence of executable statements that terminated in an error condition. At least one statement number is always printed. This number identifies the erroneous statement or, in certain cases, the last correct statement executed prior to the error. When a statement was compiled under the N option, however, the system cannot generate meaningful statement numbers during traceback. When a statement is reached through any form of GOTO, the line number for traceback is not reset. Thus an error in such a line will give the number of the last executed line in the error traceback.

The console terminal serves as FORTRAN I/O unit 4 for both input and output. Terminal input is automatically echoed on the console printer. In addition, the run-time system monitors the keyboard continually during execution of a FORTRAN program. Typing CTRL/C at any time causes an immediate return to the OS/8 Monitor. Typing CTRL/B branches to the system traceback routine and then exits to the monitor. This traceback routine causes a printout, which is similar to the error traceback and includes the current subroutine, the line number in the next higher level subroutine from which it was called, etc. This facilitates locating infinite loops when debugging a program. The following additional special characters are recognized by the console terminal handler and processed as shown:

RUBOUT Deletes last character accepted.

CTRL/U Deletes current line of input.

CTRL/I (Tabulation) Converted to appropriate number of spaces.

CTRL/Z Signals end-of-file on input.

Tentative output files (that is, files created by the FORTRAN program) are closed automatically upon successful completion of program execution provided that either:

1. An END FILE statement referencing the file was executed. (In this case FRTS assigns a file length equal to the actual length of the file.)

2. The last operation performed on the file was a write operation. (In this case FRTS proceeds as though an END FILE statement had been executed.)

3. A DEFINE FILE statement referencing the file was executed but an END FILE statement was not executed. (In this case, upon completion of program execution, FRTS assigns a file length equal to the length specified in the DEFINE FILE statement.)

Execution of a REWIND statement does not close a tentative file, nor does it modify the tentative file length.

1.4.1 Run-Time System Error Messages

The run-time system generates two classes of error messages. Messages listed in Table 1-8 identify errors that may occur during execution of a FORTRAN program and errors that may be encountered when the run-time system is reading a loader image file into memory in preparation for execution, or accepting I/O unit specifications. Except where indicated, all run-time system errors cause full traceback and an immediate return to the monitor. Nonfatal errors cause partial traceback, sufficient to locate the error, and execution continues.

Table 1-8
Run-Time System Error Messages

Error Message	Meaning
BAD ARG	Illegal argument to library function.
CAN'T READ IT!	I/O error on reading loader image file.
CAUTION - NO DP	The present hardware configuration does not include an FPP-12 Floating-Point Processor with double-precision option. Execution continues; however, all double-precision operations default to real arithmetic (with unpredictable results), and all complex operations also produce unpredictable results.
D.F. TOO BIG	Product of number of records times number of blocks per record exceeds number of blocks in file. Note that for a random access file the length in OS/8 blocks must be no less than the number of records times the integer but must be greater than the quotient of floating-point variables per record divided by 85.

(continued on next page)

Table 1-8 (Cont.)
Run-Time System Error Messages

Error Message	Meaning
DIVIDE BY 0	Attempt to divide by zero. The resulting quotient is set to zero and execution continues.
EOF ERROR	End of file encountered on input.
FILE ERROR	Any of: a. A file specified as an existing file was not found. b. A file specified as a nonexistent file would not fit on the designated device. c. More than one nonexistent file was specified on a single device. d. File specification contained "*" as name or extension.
FILE OVERFLOW	Attempt to write outside file boundaries.
FORMAT ERROR	Illegal syntax in FORMAT statement.
FPP ERROR	Hardware error on FPP start-up.
INPUT ERROR	Illegal character received as input.
I/O ERROR	Error in reading or writing a file; tried to read from an output device; or tried to write on an input device.
MORE CORE REQUIRED	The space required for the program, the I/O device handlers, I/O buffers, and the resident Monitor exceeds the available core.
NO DEFINE FILE	Direct access I/O attempted without a DEFINE FILE statement.
NO NUMERIC SWITCH	The referenced FORTRAN I/O unit was not specified to the run-time system.
NOT A LOADER IMAGE	The first input file specified to the run-time system was not a loader image file.
OVERFLOW	Result of a computation exceeds upper bound for that class of variable. The result is set equal to zero and execution continues. This error is detected only if an FPP is present.
OVERLAY ERROR	Error while reading overlay.

(continued on next page)

Table 1-8 (Cont.)
Run-Time System Error Messages

Error Message	Meaning
PARENS TOO DEEP	Parentheses nested too deeply in FORMAT statement.
SYSTEM DEVICE ERROR	I/O failure on the system device.
TOO MANY HANDLERS	Too many I/O device handlers are resident in memory, or files have been defined on too many devices.
USER ERROR	Illegal subroutine call, or call to undefined subroutine. Execution continues only if the E option was requested.
UNIT ERROR	I/O unit not assigned, or incapable of executing the requested operation.

CHAPTER 2

FORTRAN IV SOURCE LANGUAGE

A FORTRAN source program consists of statements using the language elements and the syntax described in this manual. A statement performs one of the following functions:

- Causes operations such as multiplication, division, and branching to be carried out
- Specifies the type and format of data being processed
- Specifies the characteristics of the source program

FORTRAN statements are composed of keywords (that is, words that the FORTRAN compiler recognizes) that you use with elements of the language set. These elements are constants, variables and expressions. There are two basic types of FORTRAN statements: executable and nonexecutable.

Executable statements specify the action of the program; nonexecutable statements describe the characteristics and arrangement of data, editing information, statement functions, and subprograms that you may include in the program. The compilation of executable statements results in the creation of executable code. Nonexecutable statements provide information only to the compiler; they do not create executable code.

The OS/8 FORTRAN IV language generally conforms to the specifications for American National Standard FORTRAN X3.9-1966. The following enhancements are included in OS/8 FORTRAN:

- You may use any arithmetic expression as an array subscript. If the expression is not of integer type, FORTRAN converts it to integer form.
- You may use alphanumeric literals (character strings delimited by apostrophes or quotation marks) in place of Hollerith constants.
- The statement label list in an ASSIGNed GO TO statement is optional.
- The following Input/Output (I/O) statements have been added:

DEFINE FILE	Device-oriented I/O
READ (u'r) WRITE (u'r)	Unformatted Direct Access I/O

- You may use any arithmetic expression as the initial value, increment, or limit-parameter in the DO statement, or as the control parameter in the COMPUTED GO TO statement.
- OS/8 FORTRAN permits constants and expressions in the I/O lists of WRITE statements.

All FORTRAN statements are listed in Appendix B.

All FORTRAN language elements, (constants, variables, and expressions), the character set from which you may form the language elements, and the rules governing their construction and use are described in Chapters 1 through 3.

In this manual, the FORTRAN language statements are grouped into eight categories, each of which is described in a separate chapter. The name, definition, and chapter references for each statement category are given in Table 2-1.

Table 2-1
FORTRAN Statement Categories

Chapter	Category	Function
6	Assignment Statement	Assign values to named variables and array elements.
7	Specification Statement	Declare the properties of variables, arrays, and functions.
8	DATA Statements	Assign initial values to variables and array elements.
9	Control Statements	Determine order of execution of the object program and terminate its execution.
10	Subprogram Statements	Define functions and subroutines.
11	Input/Output Statements	Transfer data between internal storage and specified input/output devices.
12	FORMAT Statements	Specify formats for data on input/output.

DOCUMENTATION CONVENTIONS

The following symbols represent special nonprinting characters:

Tab character (TAB key or <CTRL/I> key combination)

Space character (SPACE bar)

SYNTAX CONVENTIONS

This manual uses the following conventions to describe FORTRAN statement syntax:

- Upper-case words and letters, as well as punctuation marks other than TAB or SPACE, are typed as they are printed in this manual.
- Lower-case words indicate value substitution. The accompanying text describes the nature of the item you will substitute, e.g., integer variable, statement label, etc.
- Double square brackets ([[ ]]) enclose optional items.
- Ellipses (...) indicate that you may repeat the preceding item or bracketed group any number of times.

For example, if the description is

```
CALL sub [[ (a[[,a]]...) ]]
```

then all of the following are correct:

```
CALL TIMER
CALL INSPCT (I,J,3.0)
CALL REGRES (A)
```

If a syntax definition is italicized or in a different type face, it is only for visual emphasis.

CHAPTER 3

CHARACTERS AND LINES

3.1 THE FORTRAN CHARACTER SET

The FORTRAN character set consists of:

- The upper-case letters A through Z
- The numerals 0 through 9
- The special characters in Table 3-1

Table 3-1
FORTRAN Special Characters

Character	Name	Character	Name
	Space	( )	Parentheses
-	Tab	,	Comma
=	Equals	.	Decimal Point
+	Plus	'	Apostrophe
-	Minus	"	Quote
*	Asterisk	$	Dollar Sign
/	Slash		

You may type other printable characters such as %, _, and @ only as part of Hollerith constants, alphanumeric literals, or comments.

3.2 ELEMENTS OF A FORTRAN PROGRAM

A FORTRAN program consists of FORTRAN statements and optional comments. You group the statements into logical units called program units (a program unit being a sequence of statements which you terminate with an optional END statement).

A program unit can be either a main program or a subprogram. One main program and possibly one or more subprograms form the executable program.

3.2.1 Statements

Statements are grouped into two general classes: executable and nonexecutable. Executable statements are the action statements of the program; nonexecutable statements describe data arrangement and data characteristics. Nonexecutable statements may also contain editing and data conversion information.

A program consists of a series of statements, written one statement to a line. (A line is a string of up to 72 characters.) If a statement is too long to fit on one line, you may continue it on up to five additional lines (called continuation lines). (For further information, see Section 3.3.4, Continuation Indicator Field.)

A statement can refer to another statement. FORTRAN refers to such a statement by an integer number (called a label) ranging from 1 to 99999. Such a statement is most often referenced for the information it may contain or so that program execution can continue at that statement.

3.2.2 Comments

Comments are lines of text that document program action, indicate program sections and processes, and provide greater ease in reading the source program listing by identifying variables.

The FORTRAN compiler ignores comments; the comments exist only so that you can document what the program is doing.

3.3 FORTRAN LINES

A FORTRAN line consists of four fields:

1. Statement Label Field
2. Continuation Indicator Field
3. Statement Field
4. Identification Field

You may skip any of these fields when entering statements, but, except for the identification field, the spaces allotted to each field must remain present. In the case of the identification field, you may type a carriage return before reaching it.

Each printing space represents a single character. The following sections describe how to enter the source program and what information is contained in each field.

3.3.1 Using a Text Editor

When creating a source program with a text editor, you type the lines on a "character-per-column" basis. You may also use the <TAB> character to format lines.

Many text editors and terminals advance the terminal print carriage to a predefined print position when you type a <TAB>. This action, however, is not related to FORTRAN's interpretation of the <TAB> character.

NOTE

The FORTRAN system interprets a <TAB> as one character, not the number of characters (up to eight) that it will print.

For example, you may format the following lines in either of the ways shown:

```
C- INITIALIZE ARRAYS     or      C     INITIALIZE ARRAYS

10- W=3                  or      10    W=3

- SEL(1)=111200022D0     or            SEL(1)=111200022D0
```

where

- - represents a <TAB>
  represents a space character

Use space characters in a FORTRAN statement to improve the legibility of a line. The compiler ignores all spaces in a statement field except those within a Hollerith constant or alphanumeric literal. Thus, GO TO and GOTO are equivalent.

The compiler also ignores a <TAB> in a statement field; it considers a <TAB> to be the same as a space. However, in the compiler-generated source listing, FORTRAN prints the character following the <TAB> at the next tab stop (located at columns 9,17,25,33, etc.).

3.3.2 Statement Label Field

A statement label is a number that FORTRAN uses to reference one statement from another statement.

A statement label (sometimes also called a statement number) consists of from one to five decimal digits ranging from 1 through 99999. Place this label in the first five positions of a statement's first line. Any source program statement that is referenced by another statement must have a statement number.

FORTRAN ignores spaces and leading zeros preceding the statement label, e.g., FORTRAN interprets each of the following lines as statement label 105:

```
105
00105
  105
```

An all-zero statement label is illegal.

You may assign statement numbers in any order; however, each statement number must be unique in the program or subprogram. In contrast, a main program and a subprogram may contain identical

statement numbers. In this case, FORTRAN understands that reference to these numbers means the numbers in the program unit in which the reference is made.

You cannot label nonexecutable statements other than FORMAT statements.

When you type a source program with a terminal, an initial <TAB> skips over the label and continuation field.

3.3.3 Comment Indicator and Comments

A comment indicator tells FORTRAN that the text on a line is a comment when you type the letter C in column one. The compiler will print the contents of that line in the source program listing; however, it ignores the line when it compiles the program.

The following are restrictions on comments:

- All comment lines must begin with the letter C in column one.
- You cannot continue comment lines; consequently each comment line must begin with a C.
- Unlike other statements, the text of a comment can begin in the second space of a line.
- Comment lines must not intervene between a statement's initial line and its continuation line (or lines), or between successive continuation lines.

3.3.4 Continuation Indicator Field

A continuation indicator tells FORTRAN that the text on that line is part of the same statement as the preceding line.

You must reserve column six of a FORTRAN line for the continuation indicator even if you do not type a continuation indicator.

FORTRAN defines any character except a space in column 6 to be a continuation indicator.

The following are rules for using continuation indicators:

- You may divide a statement into distinct lines at any point.
- You may precede the continuation indicator with space characters only; you may not precede it with a <TAB> as an initial <TAB> skips over the continuation field.
- The characters beginning in column seven of a continuation line are considered to follow the last character of the previous line as if there were no break at that point.

- You may enter no more than 5 continuation lines for one statement.
- You cannot continue comment lines.
- A comment line must not intervene between a statement's initial line and its continuation line (or lines), or between successive continuation lines.
- You cannot assign statement numbers to continuation lines.

3.3.5 Statement Field

Type the text of a FORTRAN statement in columns 7 through 72. A <TAB> may precede the statement field rather than spaces. Note that because the compiler ignores <TAB>s and spaces (except in Hollerith constants and alphanumeric literals), you can space the text of the statement for maximum legibility.

3.3.6 Identification Field

Type a sequence number or other such identifying information in columns 73-80 of any line in a FORTRAN program. FORTRAN ignores the characters in this field.

NOTE

The FORTRAN compiler ignores text in these positions. Moreover, FORTRAN does not print a warning message if you accidently type text in this field. This is sometimes the source of inexplicable errors.

You might use this feature when typing punched card input. It is seldom used with terminals.

3.4 BLANK LINES

You may insert lines consisting only of blanks, <TAB>s, or no characters anywhere in your source program except immediately preceding a continuation line. You would use a blank line to improve the readability of a source listing; the FORTRAN compiler ignores them.

3.5 LINE FORMAT SUMMARY

The fields and the columns in which they may appear are listed in Table 3-2.

Table 3-2
Field Summary

Field	Column(s)
Statement Label	1 through 5
Continuation Indicator	6
Statement	7 through 72
Identification	73 through 80

The following example shows the placement of fields (The numbers represent column numbers.):

```
                                                      7
1    67                                               3

      DIMENSION A(12),B(10,10,10),C(13,13),D(17,00000001
     121,5)
10    READ (1,10005) (A,B,C,D)                        00000002
C THE DATA IS STORED ON DECTAPE; USE THE FORTRAN RUN  03
C TIME SYSTEM TO ASSIGN LUN 1 TO DTAx:                00000004
      CALL UPDATE(A,D)                                00000005
      IF (.NOT. END) GO TO 10                         00000006
```

CHAPTER 4

FORTRAN STATEMENT COMPONENTS

4.1 INTRODUCTION

The elements of FORTRAN statements are:

- Constants

  A constant is a fixed, self-describing value.

- Variables

  A variable is a symbolic name that represents a stored value.

- Arrays

  An array is a group of variables that you may refer to individually or collectively. The individual values are called array elements. Use a symbolic name to refer to the array.

- Expressions

  An expression can be a constant, variable, array element, or function reference. It may also be a combination of those components and certain other elements (called operators). a by those components. The result of the computation is a single value.

- Function References

  A function reference is the name of a function (often followed by a list of arguments). After FORTRAN performs the computation indicated by the function definition, it substitutes the computed value in place of the function reference.

4.2 SYMBOLIC NAMES

You use symbolic names to identify certain entities within a FORTRAN program unit. Symbolic names consist of a combination of from one to six alphanumeric characters. If you use more than six characters in a symbolic name, FORTRAN reads only the first six.

The first letter of a symbolic name must be a letter. The special characters listed in Table 3-1 may not appear in symbolic names.

Examples of valid and invalid symbolic names are:

Valid	Invalid	
NUMBER	5Q	(Begins with a numeral)
K9	B.4	(Contains a special character)

Table 4-1 indicates the types of variables that FORTRAN identifies by symbolic names.

Except as specifically mentioned in this manual, you may not use the same symbolic name to identify more than one FORTRAN entity.

Each variable indicated as "Typed" in Table 4-1 has a data type. The means of specifying the data type of a name are presented in Sections 4.3 and 7.2.

Within a subprogram, you may use symbolic names as dummy arguments. A dummy argument may represent a variable, array, array element, constant, expression, or subprogram. However, all subprograms must be uniquely named.

Table 4-1
Classes of Symbolic Names

Entity	Typed
Variables	yes
Arrays	yes
Arithmetic statement functions	yes
Processor-defined functions	yes
FUNCTION subprograms	yes
SUBROUTINE subprograms	no
Common blocks	no
Block data subprograms	no

4.3 DATA TYPES

The data type of a FORTRAN element may be inherent in its construction or implied by convention; you may also declare it explicitly. The data types available in FORTRAN, and their definitions, are listed in Table 4-2.

Table 4-2
FORTRAN Data Types

Data Type	Meaning
INTEGER	A whole number.
REAL	A decimal number; it can be a whole number, a decimal fraction, or a combination of the two.
DOUBLE PRECISION	Similar to real, but with approximately twice the degree of accuracy in its representation.
COMPLEX	A pair of real values that represents a complex number; the first represents the real part of that number, the second represents the imaginary part.
LOGICAL	The logical value "true" or "false".
OCTAL	An integer number in radix 8.

An important attribute of each data type is the amount of memory FORTRAN requires to represent a value of that type. Variations on the basic types affect either the accuracy of the represented value or the allowed range of values.

A "storage unit" is the amount of storage OS/8 FORTRAN requires to store a REAL, INTEGER, or LOGICAL value. DOUBLE PRECISION and COMPLEX values occupy two storage units. In OS/8 FORTRAN, a storage unit corresponds to 3 words of memory (i.e., 36 bits).

NOTE

Section 4.5.2 discusses the standard FORTRAN defaults for REAL and INTEGER variables.

Hollerith constants and alphanumeric literals have no data type. They assume the data type of the context in which they appear. (See Section 4.4.7 for details.)

4.4 CONSTANTS

A constant represents a fixed value; that is, a constant can represent numeric values, logical values, or character strings.

4.4.1 Integer Constants

An integer constant is a whole number with no decimal point. It may have a leading sign.

The format is:

```
snn
```

where

```
nn   is a string of from 1 to 7 decimal digits
s    is an optional algebraic sign
```

In OS/8 FORTRAN, an integer constant is a whole signed or unsigned number that contains no more than seven decimal digits. Integer constants must fall within the range -2**23 to 2**23-1 (-8,388,608 to 8,338,607). When you use integer constants as subscripts, FORTRAN uses them at modulo 2**12 (4,096 decimal).

FORTRAN ignores leading zeros in integer constants.

Precede a negative integer constant by a minus symbol. A plus symbol is optional before a positive number because FORTRAN assumes an unsigned constant to be positive; thus, +27 and 27 are identical.

With the exception of a plus or minus sign, an integer constant cannot contain any character other than the numerals 0 through 9. Specifically, embedded commas and decimal points are not allowed.

Examples:

Valid Integer Constants	Invalid Integer Constants	
0	99999999999	(Too large)
-127	3.14	(Embedded decimal point)
+32123	32,767	(Embedded comma)

4.4.2 Real Constants

There are two kinds of real constants: decimal and exponential.

4.4.2.1 **Decimal Real Constants** - A decimal real constant is a string of decimal digits with a decimal point. It may have a leading sign.

The format is:

```
s.nn
snn.nn
snn.
```

where

```
nn   is a string of numeric characters
.    is a decimal point
s    is an optional algebraic sign
```

Note that you do not always have to type a number following the decimal point, but you must always type the decimal point. The decimal point can appear anywhere in the digit string.

FORTRAN does not limit the number of digits in a decimal real constant, but only the leftmost six digits are significant. For example, in the constant 0.000012345678, all of the non-zero digits are significant (note that FORTRAN only stores 0.000012). However, in the constant 000507, the first three zeros are not significant.

You must precede a negative constant with a minus sign. The plus sign is optional preceding a positive real constant.

Except for algebraic signs and a decimal point, a real decimal constant cannot contain any character other than the numerals 0 through 9.

Examples:

Valid Real Constants	Invalid Real Constants
3.14159	1,234,567 (Embedded commas)
71712.	879877399. (Too large)
-.00127	100 (Decimal point missing)
0.0	

4.4.2.2 **Exponential Real Constants** - An exponential real constant is a decimal real constant followed by a decimal exponent.

The format is:

mmEsnn

where

mm	is an integer or real constant
nn	is a 1- to 3-digit integer constant
E	indicates that the constant is an exponential real constant
s	is an algebraic sign

An exponential real constant is a decimal number that you type in scientific notation, that is, in powers of 10. The number, nn, represents a power of 10 by which the preceding real or integer constant is to be multiplied (e.g., 1E6 represents the value 1.0 x 10**6). The magnitude of a real constant cannot be smaller than 10**-615 nor greater than 10**615.

A real constant occupies three words (i.e., six bytes) of storage. FORTRAN interprets this number as having a degree of precision slightly greater than seven decimal digits.

In OS/8 FORTRAN, an exponential real constant need not contain a decimal point.

A minus symbol must appear between the letter E and a negative exponent; a plus symbol is optional for a positive exponent.

Except for algebraic signs, a decimal point, and the letter E, a real exponential constant cannot contain any character other than the numerals 0 through 9. However, you may omit the decimal point if the number does not have a fractional part.

Examples:

Valid Real Constants	Invalid Real Constants	
2E-3	-47.E645	(Too large)
+5.0E3	325E-801	(Too small)
	5E3.2	(Decimal point misplaced)

4.4.3 Double-Precision Constants

A double-precision constant is a real or integer constant which FORTRAN stores in twice as many locations as a real constant; it thus has extra significant digits.

The format is:

```
mmDsnn
```

where

nn	is a 1- or 2-digit integer constant
D	designates a double-precision constant
s	is an optional algebraic sign
mm	is the double-precision number

A double-precision number is a number that has twice the amount of storage allocated for it in memory as a real number. A double-precision constant occupies six words (72 bits) of PDP-8 storage, and FORTRAN interprets it as a real number having a degree of precision approximately equal to 17 significant digits. FORTRAN does not limit the number of digits that precede the exponent, but only the leftmost 17 digits are significant.

Precede a negative double-precision constant by a minus symbol; a plus symbol is optional before a positive constant. Similarly, if the number is negative, a minus symbol must appear between the letter D and a negative exponent. You may omit the decimal point from a double-precision constant that does not have a fractional part.

NOTE

Double-precision arithmetic requires the presence of an FPP (Floating-Point Processor) with an extended precision option.

The magnitude of a double-precision constant cannot be smaller than 10**-615, nor greater than 10**615.

Examples:

```
1234567890D+5
+2.71828182846182D00
-72.5D-15
1D0
```

4.4.4 Complex Constants

A complex number is a number that has a real and an imaginary part.

The format is:

(rc,rc)

where

rc is a real constant

A complex constant is a pair of single-precision real constants that you separate with a comma and enclose in parentheses. The first real constant represents the real part of that number and the second represents the imaginary part. You must type the parentheses and comma as they are part of the constant. The real and imaginary parts may each be signed.

NOTE

You can only do complex arithmetic on the FPP by using the extended precision logic.

A complex constant occupies six consecutive words of storage, three for each real constant.

Examples:

(1.70391,-1.70391)
(+12739E3,0.)

4.4.5 Logical Constants

A logical constant specifies a logical value, that is, "true" or "false". Therefore, the only two logical constants possible are:

.TRUE.

and

.FALSE.

NOTE

You may abbreviate .TRUE. and .FALSE. as .T. and .F.

You must type the delimiting periods as they are part of each constant.

Only logical operators can operate on logical constants.

4.4.6 Octal Constants

An octal constant is a string of octal digits (0-7 only) preceded by the letter O.

The format is:

```
DATA/Onum/
```

where

num is an octal number
O identifies the number as an Octal constant

You may use an octal constant only in DATA statements to enter numbers in radix eight. An octal constant may be of any length, but the FORTRAN compiler uses only the 12 low-order digits.

You generally use octal constants to set bits for masking purposes.

Examples:

```
DATA JOB/O1032/
DATA BASE /O7777/
```

NOTE

The character following the first / in each of these examples is the letter O, not a zero.

4.4.7 Hollerith Constants

A Hollerith constant is a string of alphanumeric and/or special characters preceded by: (1) a number that states how many characters are in the constant, and (2) the letter H. You may use any ASCII character (including those that are not part of the FORTRAN character set).

The format is:

```
nHccc...c
```

where

n is an unsigned, non-zero integer constant indicating the number of characters in the string (including spaces and tabs)
c is any ASCII character
H identifies this as a Hollerith constant

Hollerith constants have no data type. They assume the data type of the context in which they appear.

Examples:

Valid Hollerith Constants	Invalid Hollerith Constants
16HTODAY'S DATE IS:	3HABCD (Wrong number of characters;
1H	this will be stored as ABC.)

4.4.7.1 **Alphanumeric Literals** - An alphanumeric literal is a string of ASCII characters delimited by apostrophes or quotation marks.

The format is:

```
'ccc...c'
"ccc...c"
```

where

c is a printable ASCII character; you must type both delimiting apostrophes or quotes.

An Alphanumeric literal is an alternate form of Hollerith constant. As for Hollerith constants, you may use any ASCII character (including those that are not part of the FORTRAN character set).

Alphanumeric literals have no data type. They assume the data type of the context in which they appear.

If you need to type an apostrophe within an alphanumeric literal, type it as two consecutive apostrophes.

Examples:

```
'CHANGE PRINTER PAPER TO PREPRINTED FORM NO. 721'

'TODAY''S DATE IS: '
```

You may use a quotation mark (") instead of an apostrophe. However, you may not mix quotation marks and apostrophes. Thus, the following literal is not allowed:

```
"THIS IS A MIXED LITERAL'
```

but you may type

```
"THIS ISN'T A MIXED LITERAL"
```

4.5 VARIABLES

A variable is a symbolic name that FORTRAN associates with a storage location. (The FORTRAN compiler assigns the storage locations.) The value of the variable is the value currently stored in that location; you can only change that value by assigning a new value to the variable with an assignment statement.

FORTRAN classifies variables by data type, in the same manner as constants. The data type of a variable indicates:

- The type of data it represents
- Its precision
- Its storage requirements

You may specify the data type of a variable either by type declaration statements (see Section 7.2), or by FORTRAN default typing rules (Section 4.5.2).

FORTRAN associates two or more variables with each other when each variable uses the same storage location; or, partially associates variables when part (but not all) of the storage which one variable uses is the same as part or all of the storage which another variable uses. You create associations and partial associations with:

- COMMON statements,
- EQUIVALENCE statements, and
- Actual and dummy arguments in subprogram references.

A variable is defined if the storage with which it is associated contains a datum of the same type. You can define a variable prior to program execution by typing a DATA statement or during execution by means of assignment or input statements.

Before you assign a value to a variable, it is an undefined variable, and you should not reference it except to assign a value to it. If you reference an undefined variable, an unknown value (garbage) will be obtained.

If you associate variables of differing types with the same storage location, then defining the value of one variable (for example, by assignment) causes the value of the other variable to become not defined.

4.5.1 Data Type Specification

Declaration statements (Section 7.2) associate given variables with specified data types. For example:

```
INTEGER VAR1
DOUBLE PRECISION VAR2
```

These statements indicate that FORTRAN will associate the integer variable VAR1 with a 3-word storage location and VAR2 with a 6-word double-precision storage location.

You can explicitly declare the data type of a variable only once in a program unit.

4.5.2 Default Data Types

FORTRAN assumes all variables having names beginning with I, J, K, L, M, or N represent integer data; variables having names beginning with any other letter are real variables. For example:

Real Variables	Integer Variables
ALPHA	KOUNT
BETA	ITEM
TOTAL	NTOTAL

4.6 ARRAYS

An array is a group of contiguous storage locations that you reference with a single symbolic name, the array name. You reference the individual storage locations, called array elements, by a subscript appended to the array name.

An array can have from one to seven dimensions.

The following FORTRAN statements establish arrays:

- Type declaration statements (Section 7.2)
- DIMENSION statements (Section 7.3)
- COMMON statements (Section 7.5)

Each of these statements defines:

- The name of the array
- The number of dimensions in the array
- The number of elements in each dimension

4.6.1 Array Declarations

Use an array declaration to instruct FORTRAN to reserve storage for an array.

The format is:

```
[[typ]] a(d[[,d]]...)
```

where

[[typ]]	is a data type declaration
a	is the array name
d	is a number specifying the number of elements in that part of the array

An array is a group of variables that have the same symbolic name; you address the elements of the array by means of a subscript.

Declare a variable to be an array by specifying the symbolic name that identifies the array within a program unit and indicates the properties of that array. The number of dimension declarators d indicates the number of dimensions in the array. The minimum number of dimensions is one and the maximum number is seven.

You must declare the size (i.e., the number or elements) of an array in order to reserve the needed amount of locations in which to store the array. The value of a dimension declarator specifies the number of elements in that dimension. For example, a dimension declarator value of 50, as in TABLE(50), indicates that the dimension contains 50 elements. The dimension declarators can be constant or variable.

The rules governing the dimensioning of arrays are as follows (characters enclosed within parentheses represent subscripted characters that must be either an integer variable or an integer constant):

In the equation

```
L(n)=M(1)[1+M(2)+M(2)M(3)+M(2)M(3)M(4)...M(n-1)m(n)]
```

let

```
L    = length of the entire array
n    = total number of dimensions in the array
M(i) = maximum subscript for each dimension in the array, where i
       specifies which dimension in the array is being referenced
```

In the above equation, L must not exceed 4095 in any case.

For example

```
L(1) = M(1)<4096
L(2) = M(1)[1+M(2)]<4096
L(3) = M(1)[1+M(2)+M(2)M(3)]<4096
```

In the above equation, L must not exceed 2047 when transmitting arrays, individual arrays, elements, or subportions of an array to subprograms.

For example

```
L(1) = M(1)<2047
L(2) = M(1)[1+M(2)]<2047
L(3) = M(1)[1+M(2)M(3)]<2047
```

The number of elements in an array is always equal to the product of the number of elements in each dimension. More specifically, the array IAB dimensioned as (3,4) has 24 elements (2 x 3 x 4 = 24) and takes 72 words of storage. Although FORTRAN stores arrays as a series of sequential storage locations, you may best visualize and reference arrays as if they were single- or multi-dimensional rectilinear matrices, dimensioned on a row, column, and plane basis. Thus, Figure 4-1 represents a 3-row, 3-column, 2-plane array.

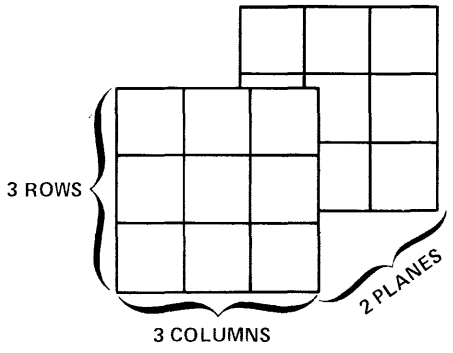


Figure 4-1 Array Representation

An array name can appear in only one declaration statement within a program unit.

Use variable dimension declarations to define adjustable arrays (see Section 4.6.5).

4.6.1.1 **Array Storage (Order of Subscript Progression)** - OS/8 FORTRAN always stores arrays in memory as a linear sequence of values. Thus, FORTRAN stores a one-dimensional array with its first element in the first storage location of the sequence and its last element in the last storage location. FORTRAN stores a multidimensional array such that the leftmost subscripts vary most rapidly. For example, in the array ARRAY(3,2,2) the progression is:

```
ARRAY(1,1,1)
ARRAY(2,1,1)
ARRAY(3,1,1)
ARRAY(1,2,1)
ARRAY(2,2,1)
ARRAY(3,2,1)
ARRAY(1,1,2)
ARRAY(2,1,2)
ARRAY(3,1,2)
ARRAY(1,2,2)
ARRAY(2,2,2)
ARRAY(3,2,2)
```

This is called the "order of subscript progression". For example, consider the following array declarators and the arrays that they create:

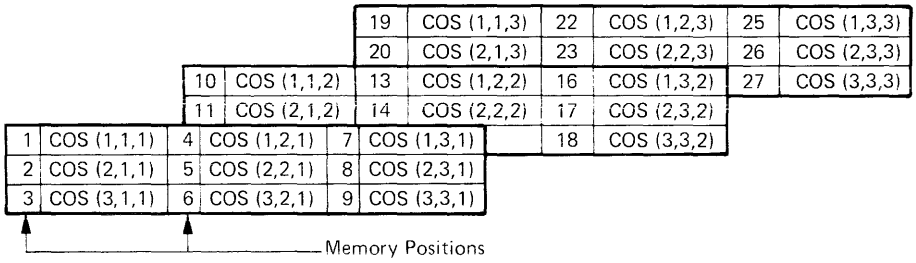


Figure 4-2 Array Storage

The arrows labeled "memory position" show the order in which FORTRAN stores information in memory. This order is critically important when you use an unsubscripted array name in a READ or WRITE statement because this is the order in which FORTRAN fills memory or prints data.

4.6.2 Subscripts

A subscript is the means by which you address individual elements in an array.

The format is:

```
(s[[,s]]...)
```

where

s is an integer subscript expression

Use a subscript following the array to specify which element in the array FORTRAN will reference.

In any subscripted array reference, you must type one subscript expression for each dimension you define for that array (i.e., one for each dimension declaration). For example, you could use the following

entry to refer to the element located in the first row, third column, second level of the array TEMP in Figure 4-2 (which is the element occupying memory position 16).

```
TEMP(1,3,2)
```

Note, however, that an array reference such as TEMP(1,3) would be illegal because the third subscript is not indicated.

Each subscript expression can be any valid integer expression. If the value of a subscript expression is not an integer, FORTRAN converts it to an integer before using it.

A subscript can be a compound expression, that is,

- Subscript quantities may contain arithmetic expressions that involve addition, subtraction, multiplication, division, and exponentiation. For example, (I+J,K*5,L/2) and (I**3,(J/4+K)*L,3) are valid subscripts.
- A subscript may contain function references. For example, TABLE(IABS(N)*KOUNT,2,3) is a valid array element identifier.
- Subscripts may contain nested array element identifiers as subscripts. For example, in the subscript (I(J(K(L)),M+N,ICOUNT), the first subscript quantity given is a nested, three-level subscript.

4.6.3 Data Type of an Array

Specify the data type of an array in the same way as the data type of a variable; that is, implicitly by the initial letter of the name, or explicitly by a type declaration statement (see Section 7.2).

All the values in an array are of the same data type. FORTRAN converts any value you assign to an array element to the data type of the array. For example, if you name an array in a DOUBLE PRECISION statement, the compiler allocates a 6-word storage location for each element of the array. When you assign a value to an element of that array, FORTRAN converts it to double precision.

4.6.4 Array References Without Subscripts

In the following type declaration statements, you may type an array name without a subscript when you wish to use the entire array.

COMMON statement

DATA statement

EQUIVALENCE statement

FUNCTION statement

SUBROUTINE statement

CALL statement

Input/Output statements

Using unsubscripted array names in any other statement is illegal.

4.6.5 Adjustable Arrays

Use an adjustable array in a subprogram so that the subprogram can process arrays of different sizes. Do this by passing the bounds as well as the array name as subprogram arguments or dummy arguments.

An adjustable array declarator, in contrast to a standard array declarator, has variable dimension declarators (which are simply integer variables). Each dimension declarator must be either an integer constant or an integer dummy argument. The array name must also appear as a dummy argument. (Consequently, you may not use adjustable array declarators in main program units.)

Upon entry to a subprogram containing adjustable array declarators, FORTRAN associates each dummy argument in a dimension declarator with an integer actual argument. FORTRAN uses these values to form the actual array declaration. These integer variables determine the size of the adjustable array for that single execution of the subprogram.

You must not change the values of the dummy adjustable array declarator arguments within the subprogram.

The effective size of the dummy array must be equal to or less than the actual size of the associated array.

The function in the following example computes the sum of the elements of a two-dimensional array. Note the use of the integer variables M and N to control the iteration.

```
      FUNCTION SUM(A,M,N)
      DIMENSION A(M,N)
      SUM = 0.
      DO 10, I = 1,M
      DO 10, J = 1,N
10    SUM = SUM + A(I,J)
      RETURN
      END
```

Following are sample calls on SUM:

```
      DIMENSION A1(10,35), A2(3,56)
      SUM1 = SUM(A1,10,35)
      SUM2 = SUM(A2,3,56)
      SUM3 = SUM(A1,10,10)
```

If there are more dimensions in the adjustable array than in the array being passed to the subroutine, you must indicate a value of 1 for that dimension declaration.

CHAPTER 5

EXPRESSIONS

5.1 INTRODUCTION

An expression is a combination of elements that represents a single value. FORTRAN relates an element in an expression to another element in the same expression by operators and parentheses. The expression can be a single basic component, such as a constant or variable, or a combination of basic components with one or more operators. Operators specify computations to be performed (using the values of the basic components) to obtain a single value.

Expressions can be classified as arithmetic, relational, or logical. Arithmetic expressions yield numeric values; relational and logical expressions produce logical values.

5.2 ARITHMETIC EXPRESSIONS

Form arithmetic expressions with arithmetic elements and arithmetic operators. The evaluation of such an expression yields a single numeric value.

An arithmetic expression element may be any of the following:

- A numeric constant
- A numeric variable
- A numeric array element
- An arithmetic expression within parentheses
- An arithmetic function reference (functions and function references are discussed in Chapter 10)

Arithmetic operators specify a computation that FORTRAN will perform using the values of arithmetic elements; they produce a numeric value as a result. The operators and their functions are listed in Table 5-1.

Table 5-1
Arithmetic Operators

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition and unary plus
-	Subtraction and unary minus

The operators listed in Table 5-1 are called binary operators, because you would use each in conjunction with two elements. You can use the + and - symbols as unary operators because, when you write them immediately preceding an arithmetic element, they indicate a positive or negative value.

5.2.1 Rules for Writing Arithmetic Expressions

Observe the following rules in structuring compound arithmetic expressions:

- An expression cannot contain two adjacent and unseparated operators. For example, the expression A*/B is not permitted.
- You must include all operators; no operation is implied. For example, the expression A(B) does not specify multiplication, although this is implied by standard algebraic notation. You must type A*(B) to obtain a multiplication of the elements.
- When you use exponentiation, the base quantity and its exponent may be of different types. For example, the expression ABC**13 involves a real base and an integer exponent. The permitted base/exponent type combinations and the type of the result of each combination are given in Table 5-2.
- You must assign a value to a variable or array element before you use it in an arithmetic expression. If you do not, the elements are undefined.

Table 5-2
Base/Exponent Combinations

BASE	EXPONENT			
	Integer	Real	Double	Complex
Integer	Yes	No	No	No
Real	Yes	Yes	Yes	No
Double	Yes	Yes	Yes	No
Complex	Yes	No	No	No

In addition, you can only exponentiate a negative element by an integer element; you cannot exponentiate an element having a value of zero by another zero-value element.

In any valid exponentiation, the result is of the same data type as the base element. The exception is a real base and a double-precision exponent; the result in this case is double precision.

5.2.2 Evaluation Hierarchy

FORTRAN evaluates arithmetic expressions in an order determined by a precedence it associates with each operator. The precedence of the operators is listed in Table 5-3.

Table 5-3
Binary Operator Evaluation Hierarchy

Operator	Precedence
**	First
* and /	Second
+ and -	Third
=	Fourth

Whenever two or more operators of equal precedence (such as + or -) appear, FORTRAN evaluates them from left to right. However, FORTRAN evaluates exponentiation from right to left. For example, A**B**C is evaluated as A**(B**C) where FORTRAN computes the parenthetical subexpression (B**C) first.

5.2.3 Data Type of an Arithmetic Expression

OS/8 FORTRAN determines the data type of an expression in the following ways:

- Integer operations - FORTRAN performs integer operations on integer elements only. (When you use octal constants and logical entities in an arithmetic context, FORTRAN treats them as integers.) In integer arithmetic, any fraction that results from a division is truncated, not rounded. For example, in integer arithmetic the value of the expression

  1/3 + 1/3 + 1/3

  is zero, not one.

- Real operations - FORTRAN performs real operations on real elements or a combination of real and integer elements. FORTRAN converts integer elements to real by giving each a fractional part equal to zero. It then evaluates the expression using real arithmetic. Note, however, that in the statement Y = (I/J)*X, FORTRAN performs an integer division operation on I and J and then performs a real multiplication on the result and X.

You can relate complex expressions only with .EQ. and .NE. operators. Complex entities are equal only if both of their corresponding real and imaginary parts are equal.

5.3 RELATIONAL EXPRESSIONS

A relational expression consists of two arithmetic expressions that you separate by a relational operator. The value of the expression is either true or false, depending on whether or not the stated relationship exists.

A relational operator tests for a relationship between two arithmetic expressions. These operators are listed in Table 5-4.

Table 5-4
Relational Operators

Operator	Relationship
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

The delimiting periods preceding and following a relational operator are part of the operator and must be present.

In a relational expression, FORTRAN evaluates the arithmetic expressions first to obtain their values. It then compares those values to determine if the relationship stated by the operator exists. For example, the expression:

```
APPLE+PEACH .GT. PEAR+ORANGE
```

tests the relationship, "The sum of the real variables APPLE and PEACH is greater than the sum of the real variables PEAR and ORANGE." If this relationship does exist, the value of the expression is true; if not, the expression is false.

All relational operators have the same precedence. Thus, if two or more relational expressions appear within an expression, FORTRAN evaluates the relational operators from left to right. Note that arithmetic operators have a higher precedence than relational operators.

Use parentheses to alter the evaluation of arithmetic expressions in a relational expression exactly as in any other arithmetic expression. However, as FORTRAN evaluates arithmetic operators before relational operators, it is unnecessary to enclose in parentheses an arithmetic expression preceding or following a relational operator.

5.4 LOGICAL EXPRESSIONS

A logical expression may be a single logical element, or it may be a combination of logical elements and logical operators. A logical expression yields a single logical value, either true or false.

A logical element can be any of the following:

- A logical constant
- A logical variable
- A logical array element
- A relational expression
- A logical expression enclosed in parentheses
- A logical function reference (functions and function references are described in Chapter 10)

The logical operators are listed in Table 5-5.

Table 5-5
Logical Operators

Operator	Example	Meaning
.AND.	A .AND. B	Logical conjunction. The expression is true if, and only if, both A and B are true.
.OR.	A .OR. B	Logical disjunction (inclusive OR). The expression is true if, and only if, either A or B, or both, is true.
.XOR.	A .XOR. B	Logical exclusive OR. The expression is true if A is true and B is false, or vice versa. It is false if both elements have the same value.
.EQV.	A .EQV. B	Logical equivalence. The expression is true if, and only if, both A and B have the same logical value, whether true or false.
.NOT.	.NOT. A	Logical negation. The expression is true if, and only if, A is false.
NOTE A and B can be expressions or constants.		

You must type the delimiting periods of logical operators.

A logical expression, like an arithmetic expression, may consist of basic elements as in

```
.TRUE.
X .GE. 3.14159
```

or

```
TVAL .AND. INDEX
BOOL(M) .OR. K .EQ. LIMIT
```

(where BOOL is either a logical function with one argument or a one-dimensional logical array).

You may enclose logical expressions within parentheses, for example,

```
A .AND. (B .OR. C)
```

or

```
(A .AND. B) .OR. C
```

Note that these expressions evaluate differently; thus, if A is false and C is true, then the first yields a false value while the second yields a true one.

5.4.1 Logical Operator Hierarchy

A summary of all operators that may appear in a logical expression, and the order in which FORTRAN evaluates them is listed in Table 5-6.

Table 5-6
Logical Operator Hierarchy

Operator	Precedence
**	First
*,/	Second
+,-	Third
Relational Operators	Fourth
.NOT.	Fifth
.AND.	Sixth
.OR.	Seventh
.XOR.,.EQV.	Eighth

5.5 USE OF PARENTHESES

In an expression, FORTRAN evaluates first all subexpressions you place within parentheses. When you nest parenthetic subexpressions (that is, one subexpression is contained within another) the most deeply nested subexpression is evaluated first, the next most deeply nested subexpression is evaluated second, and so on, until FORTRAN computes the entire parenthetical expression.

When you type more than one operation within a parenthetical subexpression, FORTRAN performs the required computations according to a hierarchy of operators (see Tables 5-4 and 5-6).

Parentheses do not imply multiplication. For example, (A+B)(C+D) is illegal.

The following example illustrates a typical numeric expression using numeric operators and a function reference. This is the familiar formula for obtaining one of the roots of a quadratic equation.

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

which might be coded

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

Note how the parentheses affect the order or evaluation. Also note that one parentheses pair is required by the SQRT function. An example of the effect of parentheses is shown below (the numbers below the operators indicate the order in which FORTRAN performs the operations).

```
4 + 3 * 2 - 6 / 2 = 7
    ^   ^   ^   ^
    2   1   4   3

(4 + 3) * 2 - 6 / 2 = 11
    ^   ^   ^   ^
    1   2   4   3

(4 + 3 * 2 - 6) / 2 = 2
    ^   ^   ^   ^
    2   1   3   4

((4 + 3) * 2 - 6) / 2 = 4
    ^   ^   ^   ^
    1   2   3   4
```

Evaluation of expressions within parentheses takes place according to the normal order of precedence.

Nonessential parentheses, such as those in the expression

```
4 + (3*2) - (6/2)
```

have no effect on the evaluation of the expression.

The use of parentheses to specify the evaluation order is often important where evaluation orders that are algebraically equivalent might not be computationally equivalent when carried out on a computer.

FORTRAN evaluates operators of equal rank from left to right.

CHAPTER 6

ASSIGNMENT STATEMENTS

6.1 INTRODUCTION

Assignment statements evaluate expressions and assign their values to variables or elements in an array.

There are three types of assignment statements:

- An arithmetic assignment statement
- A logical assignment statement
- An ASSIGN statement (see Section 9.2.3.1)

6.2 ARITHMETIC ASSIGNMENT STATEMENT

The arithmetic assignment statement assigns a numerical value to a variable or array element.

The format is:

```
v = e
```

where

```
v    is a variable or array element name
e    is an expression
```

The arithmetic assignment statement assigns the value of the expression on the right of an equal sign to the variable or array element on the left of the equal sign. If you had previously assigned a value to the variable, an assignment statement replaces it with the value on the right side of the equal sign.

Note that the equal sign does not mean "is equal to", as in mathematics. It means "is replaced by". Thus, the statement

```
KOUNT = KOUNT + 1
```

means, "Replace the current value of the integer variable KOUNT with the sum of that current value and the integer constant 1".

Although the symbolic name to the left of the equal sign can be undefined, you must previously have assigned values to all symbolic references in an expression (i.e., the right side of the equal sign).

An expression must yield a value that conforms to the requirements of the variable or array element to which you assign it. Thus, a real expression that produces a value greater than 8,338,608 is illegal if the entity on the left of the equal sign is an INTEGER variable.

If the data type of the variable or array element on the left of the equal sign is the same as that of the expression on the right, FORTRAN assigns the value directly. If the data types are different, FORTRAN converts the value of the expression to the data type of the entity on the left of the equal sign before it is assigned. A summary of data conversions on assignment is shown in Table 6-1.

Table 6-1
Conversion Rules for Assignment Statements

TO: / FROM:	REAL	INTEGER	COMPLEX	DOUBLE PRECISION	LOGICAL CONSTANT	LITERAL CONSTANT
Real	D	D	R,D	H,D	D	D,6
Integer	C	D	R,C	H,C	D	D,6
Complex	D,R,I	D,R,I	D	H,D,R,I	D,R,I	D,6
Double Precision	D,H,L	D,H,L	R,D,H,L	D	D,H,L	D,6
Logical	N	N	R,N	H,N	D	N,6

C--Conversion between integer and floating point
D--Direct replacement
H--High-order portion of expression used
I--Set imaginary part to 0
L--Set low-order part to 0
N--Convert non-zero to 1.0 (logical truth)
R--Real only (imaginary part set to 0)
6--Use the first character in the literal and five characters following

Examples:

Valid Statements

```
BETA = -1./(2.*X)+A*A/(4.*(X*X))

PI = 3.14159

SUM = SUM+1.
```

Invalid Statements

```
3.14 = A-B                  (Entity on the left must be a variable
                             or array element.)

-J = I**4                   (Entity on the left must not be signed.)

ALPHA = ((X+6)*B*B/(X-Y)    (Left and right parentheses do not
                             balance.)
```

6.3 LOGICAL ASSIGNMENT STATEMENTS

Use a logical assignment statement to assign a true or false value to a logical variable.

The format is:

```
v = e
```

where

v is a variable or array element of type logical
e is a logical expression

The logical assignment statement is similar to the arithmetic assignment statement, but it operates on logical data. The logical assignment statement evaluates the expression on the right side of an equal sign and assigns the resulting logical value, either true or false, to the variable or array element on the left.

The variable or array element on the left of the equal sign must be of type LOGICAL; its value can be undefined before the assignment.

You must have assigned values previously, either numeric or logical, to all symbolic references that appear in an expression. The expression must yield a logical value.

Examples:

```
PAGEND = .FALSE.

PRNTOK = LINE .LE. 132 .AND. .NOT. PAGEND

ABIG = A .GT. B .AND. A .GT. C .AND. A .GT. D
```

CHAPTER 7

SPECIFICATION STATEMENTS

7.1 INTRODUCTION

Specification statements in FORTRAN IV are nonexecutable statements that provide information necessary for the proper allocation and initialization of variables and names that you use in a program.

7.2 TYPE DECLARATION STATEMENTS

Type declaration statements explicitly define the data type of symbolic names.

The format is:

```
typ v[[,v]]...
```

where

typ is one of the following data type specifiers:

```
LOGICAL
INTEGER
REAL
DOUBLE PRECISION
COMPLEX
```

v is a typed variable or array

A type declaration statement causes the specified symbolic names to have the specified data type; it overrides the data type implied by the initial letter of a symbolic name.

A type declaration statement can define arrays by including array declarators (see Section 5.6.1) in the list. In each program unit, an array name can appear only once in an array declarator. Note, however, that

```
DIMENSION ISUM(?)
INTEGER ISUM
```

is legal.

Type declaration statements should precede all executable statements and all specification statements. You must precede the first use of any symbolic name with its declaration statement if you do not use the default type declaration.

You can explicitly declare the data type of a symbolic name only once.

You must not label type declaration statements. The FORTRAN entities that you may type are:

Arithmetic statement functions
Arrays
Functions
Variables

Examples:

```
INTEGER COUNT MATRIX(4,4), SUM
REAL MAN,IABS
LOGICAL SWITCH
```

7.3 DIMENSION STATEMENT

The DIMENSION statement defines the number of dimensions in an array and the number of elements in each dimension.

The format is:

```
DIMENSION a(d)[[,a(d)...]]...
```

where

a is the symbolic name of an array
d is the dimension declarator

Example:

```
DIMENSION ARRAY(6,7,4)
```

The DIMENSION statement allocates storage locations, one for each element in each dimension, for each array in the DIMENSION statement. You may declare any number of arrays in one dimension statement. Each storage location is 6 or 12 bytes in length as determined by the data type of the array. The amount of storage FORTRAN assigns to an array is equal to 6 or 12 times the product of all dimension declarators in the array declarator for that array. For example,

```
DIMENSION ARRAY(4,4), MATRIX(5,5,5)
```

defines ARRAY as having 16 real elements of 6 words each, and MATRIX as having 125 integer elements, also of 6 words each.

You cannot declare more than 7 dimensions to an array. There is also a limit of 4095 elements to any array. Each size specification must be a non-zero positive integer constant.

For further information concerning arrays and the storage of array elements, see Section 4.6.

Array declarators can also appear in type declaration and COMMON statements; however, in each program unit, an array name can appear in only one array declarator.

You must not label DIMENSION statements.

Examples:

```
DIMENSION BUD(12,24,10)

DIMENSION X(5,5,5),Y(4,85),Z(100)

DIMENSION MARK(4,4,4,4,4)
```

7.4 EXTERNAL STATEMENT

The EXTERNAL statement permits the use of external procedure names (functions, subroutines, and FORTRAN library functions) as arguments to other subprograms.

The format is:

```
EXTERNAL v[[,v]]...
```

where

v is the symbolic name of a subprogram or the name of a dummy argument associated with a subprogram

Example:

```
EXTERNAL SIN, COS, ABS
```

Any subprogram you use as an argument to another subprogram must appear in an EXTERNAL statement in the calling subprogram. Thus, the purpose of the EXTERNAL statement is to declare names to be subprogram names. This distinguishes the external name v from other variable or array names.

The subprograms may be ones that you write or those that are part of the FORTRAN library. The EXTERNAL statement declares each name v to be the name of a procedure external to the program unit. Such a name can then appear as an actual argument to a subprogram.

NOTE

If you use a complete function reference (for example, a call to the SQRT external function) in a reference such as CALL SORT(A,SQRT(B),C), the function reference is a value (the square root of B) and you do not need to define it as an external statement. You would only have to define it if you were passing the function name, i.e., CALL SORT(A,SQRT,C).

FORTRAN reserves the names you declare in an external statement throughout the compilation of the program; you cannot use them in any other declaration statement, with the exception of a type statement.

Example:

Main Program	Subprograms
EXTERNAL SIN,COS,TAN	SUBROUTINE TRIG (X,F,Y)
.	Y = F(X)
.	RETURN
CALL TRIG (ANGLE,SIN,SINE)	END
.	
.	
CALL TRIG (ANGLE,COS,COSINE)	
.	
.	FUNCTION TAN (X)
CALL TRIG (ANGLE,TAN,TANGNT)	TAN = SIN(X) / COS(X)
.	RETURN
.	END

The CALL statements pass the name of a function to the subroutine TRIG. The function is subsequently invoked by the function reference F(X) in the second statement of TRIG. Thus, the second statement becomes in effect:

```
Y = SIN(X)
Y = COS(X)
Y = TAN(X)
```

depending upon which CALL statement invoked TRIG. The functions SIN and COS are examples of trigonometric functions supplied in the FORTRAN Library.

7.5 COMMON STATEMENT

You use a COMMON statement so that a program and/or subprograms can share information.

The format is:

```
COMMON [[ /[[cb]] /]] nlist /[[cb]]/ nlist]]...
```

where

cb — is a symbolic name or is blank. If the first cb is blank, you can omit the first pair of slashes

nlist — is a list of variable names, array names, and array declarators separated by commas

Example:

```
COMMON /AREA1/A,B //C,D
```

The COMMON statement enables you to establish storage that two or more programs and/or subprograms may share and to name the variables and arrays that will occupy the common storage. The use of common storage conserves storage and provides a means to implicitly transfer arguments between a calling program and a subprogram. The transfer is implicit because no actual tranferral takes place; instead, the program unit references the common storage area.

FORTRAN determines the length of a COMMON block by the number of components and the amount of storage each component requires. COMMON blocks may be of any length, subject to the limitations of available memory.

After each common name cb, nlist lists the names of the variables and arrays that will occupy the common area cb. FORTRAN places the items for a common within common storage area in the order in which you list them in the COMMON statement or statements.

Elements you place into common storage in one program unit should agree in data type with elements referenced in a second. This is because assignment of storage is on a storage unit-for-storage unit basis, not variable-for-variable.

You may label COMMON storage areas or leave them blank (unlabeled). If you choose to label, type a symbolic name within slashes immediately before the list of items that will occupy the cb area.

For example, the statement

```
COMMON/AREA1/A,B,C/AREA2/TAB(13,3,3)
```

establishes two labeled common areas (AREA1 and AREA2).

If you are declaring a common storage area to be blank common, then you may omit the double slashes (// if and only if it is the first declaration of any common statement. Unlabeled common area is called "blank common". If the blank common declaration is not the first declaration in a COMMON statement, then the double slashes are mandatory.

For example, the statement

```
COMMON/AREA1/A,B,C//TAB(3,3,3)
```

establishes one labeled area (AREA1) and one unlabeled common area.

A given labeled common name may appear more than once in the same COMMON statement and in more than one COMMON statement within the same program or subprogram.

During compilation of a source program, FORTRAN will bring together all items you list for each labeled and blank common area in the order in which the items appear in the source program statements.

For example, the series of source program statements

```
COMMON/ST1/A,B,C/ST1/TAB(2,2)//C,D,E
.
.
COMMON/ST1/TST(3,4)//M,N
.
.
COMMON/ST2/X,Y,Z//O,P,Q
```

has the same effect as the single statement

```
COMMON/ST1/A,B,C,TST(3,4)/ST2/TAB(2,2),X,Y,Z//C,D,E,M,N,O,P,Q
```

FORTRAN treats each labeled common area as a separate, specific storage area. You assign initial values to the contents of a common area -- that is, variables and arrays -- by DATA statements in a BLOCK DATA subprogram. Declarations of a given common area in different subprograms must contain the same number, size, and order of variables and arrays as the reference array.

Common block names must be unique with respect to all subroutine and function names.

The largest definition of a given common area must be loaded first.

Storage allocation for blocks of the same name begins at the same location for all program units FORTRAN executes together. For example, if a program contains

```
COMMON A,B,C/R/X,Y,Z
```

as its first COMMON statement, and a subprogram has

```
COMMON /R/U,V,W //D,E,F
```

as its first COMMON statement, the values represented by X and U are stored in the same location. A similar correspondence holds for A and D in blank common.

If one program unit references a part of a common block, then you must use dummy variables to establish the proper correspondence. For example, if you declare a common block to contain

```
A,B,C,D,E,F,G,H,I,J,K
```

and a subprogram wishes to reference the storage location indicated by K, then you must declare a common block as in the following subprogram

```
COMMON A,B,C,D,E,F,G,H,I,J,K
```

The declaration COMMON K in the subprogram would cause a correspondence between variable A in the main program and variable K in the subprogram. (Note that any other sequence of variable names would also be correct.)

Instead of declaring each variable contained in the COMMON block, you may substitute a dummy array (provided that you are careful to match up proper storage lengths). For example, the following declaration

```
DOUBLE PRECISION DUMMY(5)
COMMON DUMMY,K
```

(where DUMMY is an arbitrary variable name) is equivalent to the statement in the preceding example.

7.5.1 COMMON Statements with Array Declarators

You may also define an array in a COMMON statement. You may not otherwise subscript array names. Also, you cannot assign individual array elements to COMMON.

7.6 EQUIVALENCE STATEMENT

You use an EQUIVALENCE statement to associate different variables with the same storage.

The format is:

```
EQUIVALENCE (nlist) [[,(nlist)]]...
```

where

nlist — is a list of variables and array elements, separated by commas. At least two components must be present in each list.

Example:

```
EQUIVALENCE (A,B),(C,D(16),E,F)
```

The EQUIVALENCE statement declares two or more entities to be associated (either totally or partially) with the same storage location.

NOTE

EQUIVALENCE associates different variable names with the same storage area in a program unit. COMMON may also associate different variable names with the same storage area, but it always makes the association between program units.

The EQUIVALENCE statement causes FORTRAN to allocate the same storage locations for all the variables or array elements contained in one parenthesized list. Note that any REAL variable made equivalent to a DOUBLE PRECISION variable shares storage with the high-order word of that variable. Mixing of data types in this way is permissible. Also, multiple components of one data type can share the storage of a single component of a higher-ranked data type. For example, in the statement

```
COMPLEX COMPLX
DIMENSION ARRAY(2)
EQUIVALENCE (COMPLX,ARRAY(1))
```

the EQUIVALENCE statement causes the two elements of the array ARRAY to occupy the same storage as the complex variable COMPLX. In this example, ARRAY(1) shares storage with the real component of COMPLX while ARRAY(2) shares storage with the imaginary part.

You can also use the EQUIVALENCE statement to equate variable names. For example, the statement

```
EQUIVALENCE (FLTLEN, FLENTH, FLIGHT)
```

causes FLTLEN, FLENTH, and FLIGHT to have the same value, provided they are also of the same data type.

An EQUIVALENCE statement in a subprogram must not contain dummy arguments.

Examples:

```
EQUIVALENCE (A,B), (B,C)        (has the same effect as EQUIVALENCE
                                (A,B,C))

EQUIVALENCE (A(1),X), (A(2),Y), (A(3),Z)
```

7.6.1 Making Arrays Equivalent

When you make an element of an array equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between other elements of the two arrays. Thus, if you make the first elements of two equal-sized arrays equivalent, both arrays share the same storage space. Moreover, if you make the third element of a five-element array equivalent to the first element of another array, the last three elements of the first array overlap the first three elements of the second array.

The EQUIVALENCE statement must not attempt to assign the same storage location to two or more elements of the same array, nor to assign memory locations in any way that is inconsistent with the normal linear storage of array elements (e.g., making the first element of an array equivalent with the first element of another array, then attempting to set an equivalence between the second element of the first array and the sixth element of the other).

In the EQUIVALENCE statement only, it is possible to identify an array element with a single subscript (i.e., the linear element number), even though you have defined one as being multidimensional.

For example, the statements:

```
DIMENSION TABLE (2,2), TRIPLE (2,2,2)
EQUIVALENCE (TABLE(4), TRIPLE(7))
```

result in the entire array TABLE sharing a portion of the storage space FORTRAN allocates to array TRIPLE as illustrated in Figure 7-1. In Figure 7-1, the elements with asterisks are those explicitly mentioned in the above EQUIVALENCE statement.

Array TRIPLE		Array TABLE	
Array Element	Element Number	Array Element	Element Number
TRIPLE(1,1,1)	1		
TRIPLE(2,1,1)	2		
TRIPLE(1,2,1)	3		
TRIPLE(2,2,1)	4	TABLE(1,1)	1
TRIPLE(1,1,2)	5	TABLE(2,1)	2
TRIPLE(2,1,2)	6	TABLE(1,2)	3
TRIPLE(1,2,2)	7*	TABLE(2,2)	4*
TRIPLE(2,2,2)	8		

Figure 7-1 Equivalence of Array Storage

Figure 7-1 also illustrates that the two statements

```
EQUIVALENCE (TABLE(1),TRIPLE(4))
EQUIVALENCE (TRIPLE(1,2,2), TABLE(4))
```

result in the same alignment of the two arrays.

7.6.2 EQUIVALENCE and COMMON Interaction

When you make components equivalent to entities in common, it can cause FORTRAN to extend the common block beyond its original boundaries.

An EQUIVALENCE statement can only extend common beyond the last element of the previously established common block. It must not attempt to increase the size of common in such a way as to place the extended portion before the first element of existing common. For example:

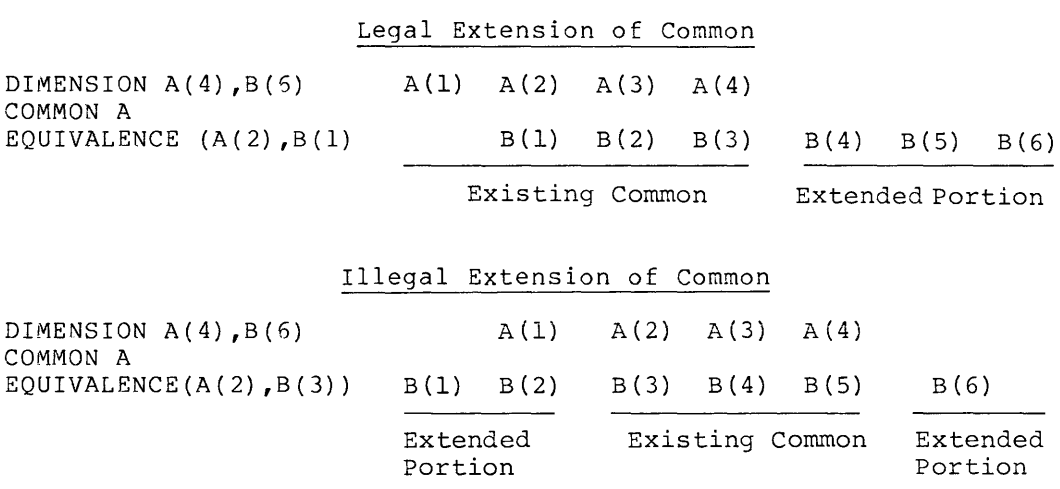


Figure 7-2 Legal and Illegal Common Extensions

If you assign two components to the same or different common blocks, you must not make them equivalent to each other.

CHAPTER 8

DATA STATEMENTS AND BLOCK DATA SUBPROGRAMS

8.1 DATA STATEMENTS

The DATA initialization statement permits the assignment of initial values to variables and array elements prior to program execution.

The format is:

```
DATA nlist/clist/[[ [[,]]nlist/clist/]]...
```

where

nlist	is a list of one or more variable names, array names, or array element names separated by commas
,	is an optional separator
clist	is a list of constants

Example:

```
DATA A,B,C(3),C(7)/4.0,8.1,16.0,28.0/
```

The DATA statement causes FORTRAN to assign the constant values in each clist to the entities in the preceding nlist. FORTRAN assigns values in a one-to-one manner in the order in which they appear, from left to right.

When an unsubscripted array name appears in a DATA statement, FORTRAN assigns values to every element of that array. The associated constant list must therefore contain enough values to fill the array. FORTRAN fills array elements in the order of subscript progression (see Section 4.6.1).

When you assign Hollerith data to a variable or array element, the number of characters that you can assign depends on the data type of the component. If the number of characters in a Hollerith constant or alphanumeric literal is less than the capacity of the variable or array element, the constant is padded on the right with spaces. If the number of characters in the constant is greater than the maximum number that the variable can hold, it ignores the rightmost excess characters.

When you assign the same value to more than one item in nlist, you may use a repeat specification. Write the repeat specification as N*D where N is an integer that specifies how many times the value of item D is to be used. For example, a DATA specification of /3*20/ specifies that the value 20 is to be assigned to the first three items named in the preceding list. Also, the statement

```
DATA M,N,L /3*20/
```

assigns the value 20 to the variables M, N, and L. The number of constants in a constant list must correspond exactly to the number of entities specified in the preceding name list. The data types of the data elements and their corresponding symbolic names must agree.

FORTRAN IV converts the constant to the type of the variable being initialized.

Example:

```
INTEGER A(10),BELL,K(5,5,5)
DATA A,BELL,STARS/10*0,7, '****'/K/25*0,25*1,25*2,25*3,25*4,25*5/
```

The DATA statement assigns zero to all ten elements of array A, the value 7 to the variable BELL, and four asterisks to the real variable STARS. The 125-element array, K, is initialized so that each of the five planes (i.e., the third dimension declarator) has a different value.

When you initialize an array, you must initialize the entire array. Thus, the DATA statement in the example

```
DIMENSION K
DATA K /10*1/
```

is illegal.

You could make the DATA statement of the example legal as follows:

```
DIMENSION I(30),K(10)
EQUIVALENCE (I,K)
DATA K/10*1/
```

The values you assign with a DATA statement may also be assigned with a BLOCK DATA subprogram. However, note that initial values for variables in COMMON storage may not be specified in subprograms that may be overlaid at execution time. If a subprogram will be overlaid, then you should only initialize these variables in a BLOCK DATA subprogram. (It is good programming practice to use BLOCK DATA subprograms to initialize only variables in COMMON storage.)

8.2 BLOCK DATA SUBPROGRAM

You use a BLOCK DATA to initialize variables you place into COMMON storage.

The format is:

```
BLOCK DATA
```

Use the BLOCK DATA subprogram to assign initial values to entities in common blocks, at the same time establishing and defining those blocks. The subprogram consists of a BLOCK DATA statement followed by a series of specification statements.

The statements FORTRAN allows in a BLOCK DATA subprogram are:

Type declaration
DIMENSION
COMMON
EQUIVALENCE
DATA

The specification statements in the BLOCK DATA subprogram establish and define common blocks, assign variables and arrays to those blocks, and assign initial values to those components.

A BLOCK DATA statement must be the first statement of a BLOCK DATA subprogram. You must not label the BLOCK DATA statement.

A BLOCK DATA subprogram must not contain any executable statements.

If you initialize any entity in a common block in a BLOCK DATA subprogram, you must enter a complete set of specification statements to establish the entire block, even though some of the components in the block do not appear in a DATA statement. You can define initial values for more than one common area with the BLOCK DATA subprogram.

Example:

```
BLOCK DATA
INTEGER S, X
LOGICAL T, W
DOUBLE PRECISION U
DIMENSION R(3)
COMMON /AREA1/R,S,T,U/AREA2/W,X
DATA R /1.0,2*2.0/ T /.FALSE./  U /0.214537D-7/  W /.TRUE./
END
```

CHAPTER 9

CONTROL STATEMENTS

9.1 INTRODUCTION

FORTRAN normally executes statements in the order in which you write them. However, it is frequently desirable to change the normal program flow by transferring control to another section of the program or to a subprogram. Transfer of control from a given point in the program may occur every time that point is reached in the program flow, or may be based on a decision made at that point.

Transfer of control, whether within a program unit or to another program unit, is effected by control statements. These statements also govern iterative processing, suspension of program execution, and program termination. The types of control statements discussed in this chapter are:

ASSIGN	IF
CONTINUE	GO TO
DO	PAUSE
END	STOP

A second kind of statement for transferring control, subprograms, is discussed in Chapter 10.

9.2 GOTO STATEMENTS

GOTO statements transfer control within a program unit, either to the same statement every time or to one of a set of statements, based on the value of an expression.

The three types of GOTO statements are:

- Unconditional
- Computed
- Assigned

9.2.1 Unconditional GOTO Statement

This type of GOTO statement transfers control to the same statement every time it is executed.

The format is:

```
GOTO st
```

where

st is the label of an executable statement in the same program unit as the GOTO statement

Example:

```
GOTO 50
```

The unconditional GOTO statement transfers control to the statement identified by the specified label. The statement label must identify an executable statement in the same program unit as the GOTO statement.

Examples:

```
GOTO 7734

GOTO 99999

GOTO 27.5        (Invalid; the statement label is improperly formed.)
```

9.2.2 Computed GOTO Statement

This type of GOTO statement transfers control to a statement based on the value of an expression within the statement.

The format is:

```
GOTO (slist)[[,]] e
```

where

slist is a list of one or more executable statement labels separated by commas
, is an optional separator
e is an integer expression the value of which falls within the range 1 to n (where n is the number of statement labels in slist)

Example:

```
GOTO (10,200,25), NUMBER
```

Use the computed GOTO to transfer control to one statement out of a list of statements. The computed GOTO thus acts as a multidirectional switch.

The computed GOTO statement evaluates the integer expression e and then transfers control to the e'th statement label in slist. That is, if the list contains (30,20,30,40), and the value of e is 2, the GOTO statement transfers control to statement 20, and so on.

You may include any number of statements in slist, but you must use each number as a label within the program.

Examples:

```
GOTO (12,24,36),INCHES

GOTO (320,330,340,350,360)ISITU(J,K)+1
```

If the value of the expression is less than 1, or greater than the number of labels in the slist, unpredictable results occur.

9.2.3 ASSIGN and ASSIGNed GOTO Statement

9.2.3.1 **ASSIGN Statement** - You use the ASSIGN statement to assign a statement label to a variable name.

The format is:

```
ASSIGN st to v
```

where

- st is the label of an executable statement in the same program unit as the ASSIGN statement
- v is an integer variable

Example:

```
ASSIGN 50 TO NUMBER
```

Use the ASSIGN statement to associate a statement label with an integer variable. You can then use the variable as a transfer destination in a subsequent ASSIGNed GOTO statement (see Section 9.2.3.2).

NOTE

The statement number must be in the same program unit.

The statement label st must not be the label of a FORMAT statement.

The ASSIGN statement assigns the statement number to the variable in a manner similar to that of an arithmetic assignment statement, with one exception: the variable becomes defined for use as a statement label reference and becomes undefined as an integer variable.

FORTRAN must execute an ASSIGN statement before the ASSIGNed GOTO statement in which it will use the assigned variable. The ASSIGN statement and the ASSIGNed GOTO statement must occur in the same program unit.

For example, the statement

```
ASSIGN 100 TO NUMBER
```

associates the variable NUMBER with the statement label 100.

Arithmetic operations on the variable, as in the statement

```
NUMBER = NUMBER + 1
```

then become invalid, because FORTRAN cannot alter a statement label. (This is because a statement refers to a location in memory and is not a number.) The statement

```
NUMBER = 10
```

disassociates NUMBER from statement 100, assigns it an integer value 10, and returns it to its status as an integer variable. After you make such an assignment, you can no longer use it in an ASSIGNed GOTO statement.

Examples:

```
ASSIGN 10 TO NSTART

ASSIGN 99999 TO KSTOP

ASSIGN 250 TO ERROR        (You must first define ERROR as an
                           integer variable.)
```

9.2.3.2 **ASSIGNed GOTO Statement** - The ASSIGNed GOTO transfers control to a statement that is represented by a variable.

The format is:

```
GOTO v[[ [[,]](slist)]]
```

where

v	is an integer variable
,	is an optional separator
slist	(when present) is a list of one or more executable statement labels separated by commas

Example:

```
GOTO NUMBER,(10,35,15)
```

The ASSIGNed GOTO statement transfers control to the statement whose label was most recently assigned to the variable v by an ASSIGN statement.

The variable v must be of integer type. In addition, you must have previously assigned to it a statement label number with an ASSIGN statement (not an arithmetic assignment statement).

The ASSIGNed GOTO statement and its associated ASSIGN statement must reside in the same program unit. Also, statements to which FORTRAN transfers control must be executable statements in the same program unit.

Examples:

```
ASSIGN 50 TO IGO
GOTO IGO

GOTO INDEX, (300,450,1000,25)
```

If the statement label value of v is not present in the list slist (and a list is specified), control transfers to the next executable statement following the ASSIGNed GOTO statement.

NOTE

You must label the statement following an ASSIGNed GOTO; otherwise, FORTRAN can never execute that statement.

9.3 IF STATEMENTS

An IF statement causes a conditional control transfer or the conditional execution of a statement. There are two types of IF statements:

- Arithmetic IF statements
- Logical IF statements

9.3.1 Arithmetic IF Statement

You use the arithmetic IF as a three-way branching statement. The branching depends on whether the value of an expression is less than, equal to, or greater than zero.

The format is:

```
IF (e) st1, st2, st3
```

where

e	is an arithmetic expression
st1, st2, st3	are the labels of executable statements in the same program unit

Example:

```
IF (I-K) 10, 20, 30
```

Use the arithmetic IF statement for conditional control transfers. This statement can transfer control to one of three statements, based on the value of an arithmetic expression.

You may use logical expressions in arithmetic IF statements. In such a case, FORTRAN first converts the logical expression value to an integer. If you use a complex expression, FORTRAN only uses the real portion.

Normal use of the arithmetic IF requires that all three labels, st1, st2, and st3, must be present. However, they need not refer to three different statements. If desired, one or two labels can refer to the same statement.

OS/8 FORTRAN allows you to type less than three numbers. If you type either one or two numbers, control passes to the next statement when a condition is not met (e.g., e is greater than zero).

Example:

```
IF (ALPHA) 10
STOP
```

In this statement, control transfers to statement number 10 if ALPHA is negative. If ALPHA is positive or equal to zero, execution stops.

The arithmetic IF statement first evaluates the expression in parentheses and then transfers control to one of the three statement labels that follow expression e. The values according to which FORTRAN makes the selection are listed in Table 9-1.

Table 9-1
Arithmetic IF Transfers

If the Value Is:	Control Passes To:
Less than 0	Label st1
Equal to 0	Label st2
Greater than 0	Label st3

Examples:

```
IF (THETA-CHI) 50,50,100
```

This statement transfers control to statement 50 if the real variable THETA is less than or equal to the real variable CHI. Control passes to statement 100 only if THETA is greater than CHI.

```
IF (NUMBER/2*2-NUMBER) 20,40
```

This statement transfers control to statement 40 if the value of the integer variable NUMBER is even, and to statement 20 if it is odd.

9.3.2 Logical IF Statement

You use a logical IF statement for conditional execution of statements.

The format is:

```
IF (e) st
```

where

e is a logical expression
st is a complete FORTRAN statement. The statement can be any executable statement except a DO statement or another logical IF statement.

Example:

```
IF(X .EQ. Y) Z=4
```

FORTRAN bases the decision to execute the conditional statement on the value of a logical expression within the statement.

The logical IF statement first evaluates the logical expression. If the value of the expression is true, FORTRAN transfers control to the executable statement within the IF statement. If the value of the expression is false, control transfers to the next executable statement following the logical IF; in this case, FORTRAN does not execute statement st.

Examples:

```
IF (J .GT. 4 .OR. J .LT. 1) GOTO 250

IF (REF(J,K) .NE. HOLD) REF(J,K) = REF(J,K)*A(K,J)

IF (.NOT. X) CALL SWITCH(S,Y)
```

9.4 DO STATEMENT

You use the DO statement to execute a block of statements repeatedly.

The format is:

```
DO st i=e1,e2[[,e3]]
```

where

- st is the label of an executable statement that physically follows in the same program unit
- i is an unsubscripted real or integer variable
- e1 (the initial value of i) is an integer, real constant, or expression
- e2 (the terminal value of i) is an integer, real constant, or expression and must be greater than e1
- e3 (the value by which i will be incremented each time it executes the statements in the range of the DO loops) is an integer, real constant, or expression

Example:

```
DO 10 I=1,10,2

DO 20 I=J,K,L
```

The DO statement causes FORTRAN to execute the statements in its range a specified number of times.

The range of a DO statement is defined as the series of statements that follow the DO statement up to and including its specified terminal statement st; that is, the statements that follow the DO statement, up to and including the terminal statement, are in the range of the DO loop.

The variable i is called the control (or index) variable of the DO and e1, e2, e3 are the initial, terminal, and increment parameters respectively.

The terminal statement of a DO loop is identified by the label st that appears in the DO statement. This terminal statement must not be a GOTO statement, an arithmetic IF statement, a RETURN statement, a PAUSE statement, a STOP statement, or another DO statement. A logical IF statement is acceptable, provided it does not contain any of the above statements.

The DO statement first evaluates the expressions e1, e2, e3 to determine values for the initial, terminal, and increment parameters. FORTRAN then assigns the value of the initial parameter to the control variable. FORTRAN then repeatedly executes the statements in the range of the DO loop.

The increment parameter must be positive and not zero; the value of the terminal parameter must not be less than that of the initial parameter.

After each execution of the range of the DO loop, FORTRAN adds the increment value to the value of the index. It then compares the result to the terminal value. If the index value is not greater than the terminal value, FORTRAN reexecutes the range using the new value of the index i.

The number of executions of the DO range, called the iteration count, is given by

```
MAX(1,((e2-e1)/e3) + 1
```

FORTRAN always executes the range of a DO statement at least once.

9.4.1 DO Iteration Control

You can terminate the execution of a DO by a statement within the range that transfers control outside the loop. When you transfer out of the DO loop's range, the control variable of the DO remains defined with its current value.

When execution of a DO loop terminates, if other DO loops share the same terminal statement, control transfers outward to the next most enclosing DO loop in the DO nesting structure (Section 9.4.2). If no other DO loops share this terminal statement, or if this DO is the outermost DO, control transfers to the first executable statement following the terminal statement.

You may alter the values of i, e1, e2, and e3. If you alter the value of i, the loop will not be executed the number of times that you originally specified. If you alter the values of the expressions, you do not affect the looping because FORTRAN "remembers" these values. The control variable i is available for reference as a variable within the range.

The range of a DO loop can contain other DO statements, so long as those "nested" DO loops conform to certain requirements (see Section 9.4.2).

Although you can transfer control out of a DO loop, you cannot transfer into a loop from elsewhere in the program. Exceptions to this rule are described in the following sections.

Examples:

```
DO 100 K=1,50,2      (25 iterations, K=49 during final iteration)

DO 25 IVAR=1,5       (5 iterations, IVAR=5 during final iteration)

DO NUMBER=5,40,4     (Invalid; statement label missing)

DO 40 M=2.10         (Invalid; decimal point instead of comma)
```

The last example illustrates a common clerical error. It is a valid arithmetic assignment statement in the FORTRAN language; i.e.,

```
DO40M = 2.10
```

9.4.2 Nested DO Loops

A DO loop may contain one or more complete DO loops. The range of an inner-nested DO must lie completely within the range of the next outer loop. Nested loops may share the same terminal statement.

Correctly Nested DO Loops

```
    DO 45 K=1,10
     .
     .
    DO 35 L=2,50,2
     .
     .
 35 CONTINUE
     .
     .
    DO 45 M=1,20
     .
     .
 45 CONTINUE
```

Incorrectly Nested DO Loops

```
    DO 15 K=1,10
     .
     .
    DO 25 L=1,20
     .
     .
 15 CONTINUE
     .
     .
    DO 30 M=1,15
     .
     .
     .
 25 CONTINUE
     .
     .
 30 CONTINUE
```

Figure 9-1 Nesting of DO Loops

In the correctly nested DO loops, note that the diagrammed lines do not cross. They do, however, share the same statement (45). In the incorrectly nested DO loops, the loop defined by DO 25 crosses the ranges of the other two DO loops.

Note that you may nest loops to a depth of (at least) 10 levels.

9.4.3 Control Transfers in DO Loops

Within a nested DO loop structure, you can transfer control from an inner loop to an outer loop. A transfer from an outer loop to an inner loop is illegal.

If two or more nested DO loops share the same terminal statement, you can transfer control to that statement only from within the range of the innermost loop, that is, the terminal statement belongs solely to the innermost DO statement. Any other transfer to that statement constitutes a transfer from an outer loop to an inner loop because the shared statement is part of the range of the innermost loop.

The following rules govern the transfer of program control from within the DO statements range or the ranges of nested DO statements.

- FORTRAN permits a transfer out of the range of any DO statement at any time. When such a transfer executes, the controlling DO statement's index variable retains its current value.
- FORTRAN permits a transfer into the range of a DO statement from within the range of any: DO loop; nested DO loop; or extended range loop (in which you leave the loop via a GOTO, execute statements elsewhere, and return to the original loop).

9.4.4 Extended Range

A DO loop is said to have an extended range if it contains a control statement that transfers control out of the loop and if, after the execution of one or more statements, another control statement returns control back into the loop. In this way, FORTRAN extends the range of the loop to include all of the executable statements between the destination statement of the first transfer and the statement that returns control to the loop.

Figure 9-2 illustrates valid and invalid control transfers.

Valid Control Transfers

```
        DO 35 K=1,10
         .
        DO 15 L=2,20
         .
        GOTO 20
         .
15      CONTINUE
         .
20      A=B+C
         .
        DO 35 M=1,15
         .
        GO TO 50
         .
30      X=A*D
         .
35      CONTINUE
         .
         .
50      D=E/F
         .
         .
        GOTO 30
```

Extended Range

Invalid Control Transfers

```
        GOTO 20
         .
        DO 50 K=1,10
         .
20      A=B+C
         .
        DO 35 L=2,20
         .
30      D=E/F
         .
35      CONTINUE
         .
        GO TO 40
         .
        DO 45 M=1,15
         .
40      X=A*D
         .
45      CONTINUE
         .
50      CONTINUE
         .
        GOTO 30
```

Figure 9-2 Control Transfers and Extended Range

The following rules govern the use of a DO statement extended range.

- The statement you want to transfer out of an extended range operation must be within the most deeply nested DO statement that contains the location to which the return transfer is to be made.
- You may transfer into the range of a DO statement only from the extended range of that DO statement.
- You may not use another DO statement in the extended range of a DO statement.
- The extended range of a DO statement cannot change the index variable or indexing parameters of the DO statement.
- You may execute subprograms within an extended range.

9.5 CONTINUE STATEMENT

Insert a CONTINUE statement where you do not wish a statement to be executed.

The format is:

```
st CONTINUE
```

where

st is a statement label

A CONTINUE statement is a statement that holds a place in the program without performing any operations.

You may place CONTINUE statements anywhere in the source program without affecting the program sequence of execution. CONTINUE statements are commonly used as the last statement of a DO statement range in order to avoid ending with a GOTO, PAUSE, STOP, RETURN, arithmetic IF, another DO statement, or a logical IF statement containing one of the previous statements.

Note that you also use a CONTINUE as a transfer point for a GOTO statement within the DO loop that is intended to begin another repetition of the loop.

Example:

In the following sequence, the labeled CONTINUE statement provides a legal termination for the range of the DO loop.

```
        .
        .
        DO 45 ITEM=1,1000
        STOCK=NVNTRY(ITEM)
        IF (STOCK .EQ. TALLY) GO TO 45
        CALL UPDATE(STOCK,TALLY)
        IF (ITEM .EQ. LAST) GO TO 77
45      CONTINUE
        .
        .
        .
77      WRITE (4,20) HEADING, PAGENO
        .
        .
```

9.6 PAUSE STATEMENT

You use the PAUSE statement to suspend program execution temporarily to give yourself time to perform some action.

The format is:

```
PAUSE [[num]]
```

where

num is an optional integer variable or expression containing one to five digits

The PAUSE statement prints the display (if you have specified one) at your terminal, suspends program execution, and waits for you to type the RETURN key. This causes program execution to resume with the first executable statement following the PAUSE.

Examples:

```
PAUSE "13731

PAUSE 'MOUNT TAPE REEL #3'
```

9.7 STOP STATEMENT

You use the STOP statement to terminate program execution.

The format is:

```
STOP
```

When the STOP statement terminates program execution, it returns control to the operating system. If you do not type a STOP statement, a "stop" occurs when FORTRAN transfers control to an END statement in the main program unit.

A CALL EXIT statement is equivalent to STOP and closes any temporary files at the last block written on the file. Control returns to the OS/8 monitor.

Examples:

```
      STOP

99999 STOP
```

9.8 END STATEMENT

You mark the end of every program unit with an END statement, which must be the last source line of every program unit.

The format is:

```
END
```

In a main program, if control reaches the END statement, execution of the program terminates; in a subprogram, a RETURN statement is implicitly executed.

In the main program, END is equivalent to STOP; in a subprogram, it is equivalent to RETURN.

A program cannot reference an END statement.

Control returns to the OS/8 monitor after FORTRAN executes an END statement.

If you do not type an END statement as the last statement in your program, FORTRAN appends one.

CHAPTER 10

SUBPROGRAMS

10.1 INTRODUCTIONS

Procedures you use repeatedly in a program may be written once and then referenced each time you need the procedure. Procedures that you may reference are either internal (written and contained within the program in which they are referenced) or external (self-contained executable procedures that you may compile separately). The kinds of procedures that you may reference are:

- Arithmetic statement functions
- External functions
- Subroutines
- Intrinsic functions (FORTRAN-defined functions)

10.2 SUBPROGRAM ARGUMENTS

Since you may reference subprograms at more than one point throughout a program, many of the values that the subprogram uses may change each time you call the subprogram. Dummy arguments in subprograms represent the actual values that the subprogram will use. The arguments are passed to the subprogram when FORTRAN transfers control to it.

Functions and subroutines use dummy arguments to indicate the type of the actual arguments they represent and whether the actual arguments are variables, array elements, arrays, subroutine names, or the names of external functions. You must use each dummy argument within a subprogram as if it were a variable, array, array element, subroutine, or external function identifier. You enter dummy arguments in an "argument list" that you associate with the identifier assigned to the subprogram; actual arguments are normally given in an argument list that you associate with a call made to the subprogram.

The position, number, and type of each dummy argument in a subprogram must agree with the position, number, and type of each argument in the argument list of the subprogram reference.

Dummy arguments may be:

- Variables
- Array names
- Subroutine identifiers
- Function identifiers

When you reference a subprogram, FORTRAN replaces its dummy arguments with the corresponding actual arguments that you supply in the reference. All appearances of a dummy argument within a function or subroutine are related to the given actual arguments. Except for subroutine identifiers and literal constants, a valid association between dummy and actual arguments occurs only if both are of the same type; otherwise, the result of the subprogram will be unpredictable. Argument associations may be carried through more than one level of subprogram reference if a valid association is maintained through each level. The dummy/actual argument associations established when you reference a subprogram terminate when FORTRAN completes the operations defined in the subprogram.

The following rules govern the use and form of dummy arguments.

- The number and type of the dummy arguments of a procedure must be the same as the number and type of the actual arguments given each time you reference the procedure.
- You may not use dummy argument names in EQUIVALENCE, DATA, or COMMON statements.
- You should provide a variable dummy argument with a variable, an array element identifier, an expression, or a constant as its corresponding argument.
- You should provide an array dummy argument with either an array name or an array element identifier as its corresponding actual argument. If the actual argument is an array, the length of the dummy array should be less than or equal to that of the actual array. FORTRAN associates each element of a dummy array directly with the corresponding elements of the actual array.
- You must provide a dummy argument representing an external function with an external function as its actual argument.
- You should give a dummy argument representing a subroutine identifier a subroutine name as its actual argument.
- You may define (or redefine) a dummy argument in a referenced subprogram only if its corresponding actual argument is a variable. If dummy arguments are array names, then you may redefine the elements of the array.

10.3 USER-WRITTEN SUBPROGRAMS

FORTRAN transfers control to a function by means of a function reference. It transfers control to a subroutine by a CALL statement. A function reference is the name of the function, together with its arguments, appearing in an expression. A function always returns a value to the calling program. Both functions and subroutines may return additional values via assignment to their arguments. A subprogram can reference other subprograms, but it cannot, either directly or indirectly, reference itself (that is, FORTRAN is not recursive).

10.3.1 Arithmetic Statement Functions (ASF)

You use an Arithmetic statement function to define a one-statement, self-contained computational procedure.

The format is:

```
nam ([[a[[,a]]...]])=e
```

where

```
nam  is the name you assign to the ASF
a    is a dummy argument
e    is an expression
```

Examples:

```
PROOT(A,B,C) = (-B+SQRT(B**2 - 4*A*C))/(2*A)
NROOT(A,B,C) = (-B-SQRT(B**2 - 4*A*X))/(2*A)
```

An arithmetic statement function is similar in form to an arithmetic assignment statement. The appearance of a reference to the function within the same program unit causes FORTRAN to perform the computation and make the resulting value available to the expression in which the ASF reference appears.

The expression e is an arithmetic expression that defines the computation to be performed by the ASF.

You reference an ASF in the same manner as an external function.

The format is:

```
nam ([[a[,a]]...]])
```

where

```
nam  is the name of the ASF
a    is an actual argument
```

NOTE

You must define all ASFs before you type any executable statements.

When a reference to an arithmetic statement function appears in an expression, FORTRAN associates the values of the actual arguments with the dummy arguments in the ASF definition. FORTRAN then evaluates the expression in the defining statement and uses the resulting value to complete the evaluation of the expression containing the function reference.

You specify the data type of an ASF either implicitly by the initial letter of the name or explicitly in a type declaration statement.

Dummy arguments in an ASF definition only indicate the number, order, and data type of the actual arguments. You may use the same names to represent other entities elsewhere in the program unit. Note that with the exception of data type, FORTRAN does not associate declarative information (such as placement in COMMON or declaration as an array) with the ASF dummy arguments. Also, you cannot use the name of the ASF to represent any other entity within the same program unit.

The expression in an ASF definition may contain function references.

Any reference to an ASF must appear in the same program unit as the definition of that function. You cannot use an ASF name in an EXTERNAL statement.

An ASF reference must appear as, or be part of, an expression; you must not use it as the left side of an assignment statement.

Actual arguments must agree in number, order, and data type with their corresponding dummy arguments. You must assign values to actual arguments before the reference to the arithmetic statement function.

Examples:

Definitions

```
VOLUME(RADIUS) = 4.189*RADIUS**3

SINH(X) = (EXP(X)-EXP(-X))*0.5

AVG(A,B,C,3.) = (A+B+C)/3.          (Invalid; constant as dummy
                                     argument not permitted)
```

ASF References

```
AVG(A,B,C) = (A+B+C)/3.             (Definition)

     .

     .

     .

GRADE = AVG(TEST1,TEST2,XLAB)

IF (AVG(P,D,Q).LT.AVG(X,Y,Z))GOTO 300

FINAL = AVG(TEST3,TEST4,LAB2)       (Invalid; data type of third
                                     argument does not agree with dummy
                                     argument)
```

10.3.2 **FUNCTION Subprogram**

A FUNCTION is an external computing procedure that returns a value. You use this value as an expression or as part of an expression.

The format is:

```
[[typ]] FUNCTION nam(a[[,a...]])
```

where

```
typ  is an optional data type specifier
nam  is a name of the function
a    is one of a maximum of six dummy arguments
```

A FUNCTION subprogram is a program unit that consists of a FUNCTION statement followed by a series of statements that define a computing procedure. FORTRAN transfers control to a FUNCTION subprogram by a function reference and returns to the calling program unit when it encounters a RETURN statement.

You must always specify at least one argument to a FUNCTION. You may specify other arguments explicitly or place them in COMMON.

A FUNCTION subprogram returns a single value to the calling program unit by assigning that value to the function's name. FORTRAN determines the data type of the returned value by the function's name unless you have specified the data type.

A function reference that transfers control to a FUNCTION subprogram has the form:

```
nam ([[a[[,a]]...]])
```

where

```
nam  is the symbolic name of the function
a    is an actual argument
```

When FORTRAN transfers control to a function subprogram, FORTRAN associates the values you supply through the actual arguments (if any) with the dummy arguments (if any) in the FUNCTION statement. FORTRAN then executes the statements in the subprogram.

NOTE

You may not pass an array to a subprogram if it contains more than 2047 elements. You must implicitly pass larger arrays in COMMON.

You must assign a value to the name of the function before FORTRAN executes a RETURN statement in that function. When FORTRAN returns control to the calling program unit, it makes the value you have assigned to the function's name available to the expression that contains the function reference; it then uses this value to complete the evaluation of the expression.

NOTE

You can store variables that a FUNCTION requires in COMMON rather than passing them explicitly.

You may specify the type of a function name implicitly or explicitly in the FUNCTION or type declaration statement.

The FUNCTION statement must be the first statement of a function subprogram. You may not label a FUNCTION statement.

A FUNCTION subprogram must not contain a SUBROUTINE statement, a BLOCK DATA statement, or a FUNCTION statement (other than the initial statement of the subprogram). A function may, however, call another function or subroutine so long as the call is not directly or indirectly recursive.

10.3.3 SUBROUTINE Subprograms

A SUBROUTINE is an external computing procedure that you may repeatedly call from a program or subprogram.

The format is:

```
SUBROUTINE nam [[([[a[[,a]]...]])]]
```

where

nam is the name of the subroutine
a is a dummy argument

A SUBROUTINE subprogram is a program unit that consists of a SUBROUTINE statement followed by a series of statements that define a computing procedure. FORTRAN transfers control to a SUBROUTINE subprogram by a CALL statement and returns to the calling program unit by a RETURN statement.

When FORTRAN transfers control to a subroutine, it associates the values you supply with the actual arguments (if any) in the CALL statement with the corresponding dummy arguments (if any) in the SUBROUTINE statement. You may not specify more than six arguments in a subroutine call. FORTRAN then executes the statements in the subprogram.

The SUBROUTINE statement must be the first statement of a subroutine; it must not have a statement label.

A SUBROUTINE subprogram cannot contain a FUNCTION statement, a BLOCK DATA statement, or a SUBROUTINE statement (other than the initial statement of the subprogram).

Example:

```
C     MAIN PROGRAM
      COMMON NFACES, EDGE, VOLUME
      READ (4,65) NFACES, EDGE
65    FORMAT(I2,F8.5)
      CALL PLYVOL
      WRITE (4,66) VOLUME
66    FORMAT (' VOLUME=',F)
      STOP
      END

      SUBROUTINE PLYVOL
      COMMON NFACES, EDGE, VOLUME
      CUBED = EDGE**3
      GOTO (6,6,6,1,6,2,6,3,6,6,6,4,6,6,6,6,6,6,6,5,6),NFACES
1     VOLUME = CUBED * 0.11785
      RETURN
2     VOLUME = CUBED
      RETURN
3     VOLUME = CUBED * 0.47140
      RETURN
4     VOLUME = CUBED * 7.66312
      RETURN
5     VOLUME = CUBED * 2.18170
      RETURN
6     WRITE (4,100) NFACES
100   FORMAT(' NO REGULAR POLYHEDRON HAS ',I3,'FACES.')
      RETURN
      END
```

The subroutine in this example computes the volume of a regular polyhedron, given the number of faces and the length of one edge. It uses a computed GOTO statement to determine whether the polyhedron is a tetrahedron, cube, octahedron, dodecahedron, or icosahedron, and also to transfer control to the proper procedure for calculating the volume. If the number of faces of the body is other than 4, 6, 8, 12, or 20, the subroutine transmits an error message to logical unit 4 as indicated in the WRITE statement.

10.4 CALL STATEMENT

The CALL statement causes the execution of a SUBROUTINE subprogram; it can also specify an argument list for use by the subroutine.

The format is:

```
CALL s[[([[a]] [[,a]]...)]]
```

where

- s is the name of a SUBROUTINE subprogram, a user-written assembly language routine, or a DEC-supplied system subroutine, or a dummy argument associated with one of the above
- a is an actual argument

After the CALL statement has associated the values in the argument list (if the list is present) with the dummy arguments in the subroutine, it then transfers control to the first executable statement of the subroutine.

The arguments in the CALL statement must agree in number, order, and data type with the dummy arguments in the subroutine definition. They can be variables, arrays, array elements, constants, expressions, alphanumeric literals, or subprogram names (if those names have been specified in an EXTERNAL statement, as described in Section 7.4). Note that an unsubscripted array name in the argument list refers to the entire array.

Examples:

```
CALL CURVE (BASE,3.14159+X,Y,LIMIT,R(LT+2))
CALL PNTOUT (A,N,'ABCD')
```

10.5 RETURN STATEMENT

You use the RETURN statement to return control from a subprogram unit to the calling program unit.

The format is:

```
RETURN
```

When FORTRAN executes a RETURN statement in a FUNCTION subprogram, it returns control to the statement that contains the function reference. When FORTRAN executes a RETURN statement in a SUBROUTINE subprogram, it returns control to the first executable statement following the CALL statement that initiated execution of the subprogram.

A RETURN statement must not appear in a main program unit.

Example:

```
      SUBROUTINE CONVRT (N,ALPH,DATA,PRNT,K)
      DIMENSION DATA(N), PRNT(N)
      IF (N .LT. 10) GOTO 100
      DATA(K+2) = N-(N/10)*N
      N = N/10
      DATA(K+1) = N
      PRNT(K+2) = ALPH(DATA(K+2)+1)
      PRNT(K+1) = ALPH(DATA(K+1)+1)
      RETURN
100   PRNT(K+2) = ALPH(N+1)
      RETURN
      END
```

10.6 FORTRAN LIBRARY FUNCTIONS

The FORTRAN library functions are listed and described in Chapter 13. You write function references to FORTRAN library functions in the same form as function references to user-defined functions. For example,

```
R = 3.14159 * ABS(X-1)
```

causes the absolute value of X-1 to be calculated, multiplied by the constant 3.14159, and assigned to the variable R.

The data types of each library function and of the actual arguments are specified in Chapter 13. Arguments you pass to these functions may not be array names or subprogram names.

Processor-defined function references are local to the program unit in which they occur and do not affect or preclude the use of the name for any other purpose in other program units.

CHAPTER 11

INPUT/OUTPUT STATEMENTS

11.1 INTRODUCTION

You specify input of data to a program by READ statements and output by WRITE statements. You use some form of these statements in conjunction with format specifications to control translation and editing of the data between internal representation and character (readable) form.

Each READ or WRITE statement contains a reference to the logical unit to or from which data transfer is to take place. You may associate a logical unit to a device or file.

READ and WRITE statements fall into the following three categories:

- Unformatted sequential I/O transmit binary data without translation.
- Formatted sequential I/O transmit character data using format specifications to control the translation of data to characters on output, and to internal form on input.
- Unformatted direct access I/O transmit binary data without translation to and from direct access files.

To perform file management functions, you use auxiliary I/O statements. REWIND and BACKSPACE perform file positioning. The ENDFILE statement writes a special record that will cause an end-of-file condition when read by a READ statement. The BACKSPACE statement repositions a file to the previous record. The DEFINE FILE statement declares a logical unit to be connected to a direct access file and specifies the characteristics of the file.

11.1.1 Input/Output Devices and Logical Unit Numbers

OS/8 FORTRAN uses the I/O services of the operating system and thus supports all peripheral devices that are supported by the operating system. I/O statements refer to I/O devices by means of logical unit numbers, which are integer constants or variables with a positive value.

The default logical unit numbers are:

1 Paper Tape Reader
2 Paper Tape Punch
3 Line Printer
4 Terminal

The logical unit number must be in the range 1 through 9.

11.1.2 Format Specifiers

You use format specifiers in formatted I/O statements. A format specifier is the statement label of a FORMAT statement. Chapter 12 discusses FORMAT statements.

11.1.3 Input/Output Record Transmission

I/O statements transmit data in terms of records. The amount of information that one record can contain, and the way in which records are separated, depend on the medium involved.

For unformatted I/O, specify the amount of data that FORTRAN will transmit by an I/O statement. FORTRAN determines the amount of information it will transmit by the I/O statement and by specifications in the associated format specification.

If an input statement requires only part of a record, you lose the excess portion of the record in transmission. In the case of formatted sequential input or output, you may transmit one or more additional records by a single I/O statement.

11.2 INPUT/OUTPUT LISTS

An I/O list specifies the data items to be manipulated by the statement containing the list. The I/O list of an input or output statement contains the names of variables, arrays, and array elements whose values FORTRAN will transmit. In addition, the I/O list of an output statement can contain constants and expressions.

The format is:

```
s[[,s]]...
```

where

s is a simple list or an implied DO list

The I/O statement assigns input values to, or outputs values from, the list elements in the order in which they appear, from left to right.

11.2.1 Simple Lists

A simple I/O list consists of a single variable, array, array element, constant, or expression.

When an unsubscripted array name appears in an I/O list, a READ statement inputs enough data to fill every element of the array; a WRITE statement outputs all of the values contained in the array. Data transmission starts with the initial element of the array and proceeds in the order of subscript progression, with the leftmost subscript varying most rapidly. For example, if the unsubscripted name of a two-dimensional array defined as

```
DIMENSION ARRAY(3,3)
```

appears in a READ statement, that statement assigns values from the input record(s) to ARRAY(1,1), ARRAY(2,1), ARRAY(3,1), ARRAY(1,2), and so on, through ARRAY(3,3).

If, in a READ statement, you input the individual subscripts for an array, you must input the subscripts before their use in the array. If, for example, FORTRAN executes the statement

```
      READ (1,1250) J,K,ARRAY(J,K)
1250  FORMAT (I1,X,I1,X,F6.2)
```

and the input record contains the values

```
1,3,721.73
```

FORTRAN assigns the value 721.73 to ARRAY(1,3). FORTRAN assigns the first input value to J and the second to K, thereby establishing the actual subscript values for ARRAY(J,K). Variables that you use as subscripts in this way must appear to the left of their use in the array subscript.

You may use any valid expression in an output statement I/O list. However, the expression must not cause FORTRAN to attempt further I/O operations. A reference in an output statement I/O list expression to a FUNCTION subprogram that itself performs input/output is illegal.

You must not include an expression in an input statement I/O list except as a subscript expression in an array reference.

11.2.2 **Implied DO Lists**

You use an implied DO list to specify iteration within an I/O list.

The format is:

```
(list,i=e1,e2)
```

where

list	is an I/O list
i	is a control variable definition
e1	is the initial value of i
e2	is the terminal value of i

When you use an implied DO list, you may transmit only part of an array, or transmit array elements in a sequence other than the order of subscript progression. The implied DO list functions as though it were a part of an I/O statement that resides in a DO loop.

When you use nested implied DO lists, the first control variable definition is equivalent to the innermost DO of a set of nested loops, and therefore varies most rapidly. For example, the statement

```
      WRITE (5,150) ((FORM(K,L), L=1,10), K=1,10)
150   FORMAT (F10.2)
```

is similar to

```
      DO 50 K=1,10
      DO 50 L=1,10
      WRITE (5,150) FORM(K,L)
150   FORMAT (F10.2)
50    CONTINUE
```

Since the inner DO loop is executed ten times for each iteration of the outer loop, the second subscript, L, advances from one through ten for each increment of the first subscript. This is the reverse of the order of subscript progression.

The implied DO uses the control variable of the imaginary DO statement to specify which value or values are to be transmitted during each iteration of the loop.

i, e1, and e2 have the same form as that used in the DO statement. The rules for the control, initial, and terminal variables of an implied DO list are the same as those for the DO statement. Note, however, that an implied DO loop cannot use an increment parameter. The list may contain references to the control variable as long as the value of the control variable is not altered. There is no extended range for an implied DO list.

Examples:

```
WRITE (3,200) (A,B,C,  I=1,3)

WRITE (6,15) L,M,(I,(J,P(I),Q(I,J),J=1,L),I=1,M)

READ (1,75) (((ARRAY(M,N,I), I=2,8), N=2,8), M=2,8)
```

FORTRAN transmits the entire list of the implied DO before the incrementation of the control variable. For example

```
READ (3,999) (P(I), (Q(I,J), J=1,10), I=1,5)
```

assigns input values to the elements of arrays P and Q in the order:

```
P(1), Q(1,1), Q(1,2), ... , Q(1,10),
P(2), Q(2,1), Q(2,2), ... , Q(2,10),
 .       .       .           .
 .       .       .           .
P(5), Q(5,1), Q(5,2), ... , Q(5,10)
```

When processing multidimensional arrays, you may use a combination of a fixed subscript and subscript or subscripts that varies according to an implied DO. For example

```
READ (3,5555) (BOX(1,J), J=1,10)
```

assigns input values to BOX(1,1) through BOX(1,10) and then terminates without affecting any other element of the array.

It is also possible to output the value of the control variable directly, as in the statement

```
WRITE (6,1111) (I, I=-1,20)
```

which simply prints the integers one through twenty.

11.3 INPUT/OUTPUT FORMS

11.3.1 Unformatted Sequential Input/Output

Unformatted input and output is data in internal (binary) format without conversion or editing. Use unformatted I/O statements when data output by a program is to be subsequently input by the same program (or a similar program). Unformatted I/O statements save execution time because they eliminate the data conversion process, preserve greater precision in the external data, and usually conserve file storage space.

11.3.2 Formatted Sequential Input/Output

You use formatted input and output statements in conjunction with FORMAT statements to translate and edit data on output for ease of interpretation, and, on input, to convert data from external format to internal format.

11.3.3 Unformatted Direct Access Input/Output

You use unformatted direct access READ and WRITE statements to perform direct access I/O with a file on a direct access device. Use the DEFINE FILE statement to establish the number of records, and the size of each record, in a file to which FORTRAN will perform direct access I/O. Each direct access READ or WRITE statement contains an integer expression that specifies the number of the record to be accessed. The record number must not be less than one nor greater than the number of records you define for the file.

In OS/8 FORTRAN, the expression that specifies the record number can be of any type. FORTRAN converts it to integer type if necessary.

11.4 READ STATEMENTS

11.4.1 Unformatted Sequential READ Statement

You use unformatted sequential READ statements to assign fields to a record without translating stored information into external form.

The format is:

```
READ (u)[[list]]
```

where

```
u    is a logical unit number from 1 to 9
list is an I/O list
```

The unformatted sequential READ statement inputs one unformatted record from a logical unit and assigns the fields of the record without translation to the I/O list elements in the order in which they appear, from left to right.

An unformatted sequential READ statement transmits exactly one record. If the I/O list does not use all of the values in the record, FORTRAN discards the remainder of the record. If FORTRAN exhausts the contents of the record before the I/O list is satisfied, an error condition results.

You must only use the unformatted sequential READ statement to read records that were created by unformatted sequential WRITE statements.

If you use an unformatted WRITE statement that does not contain an I/O list, FORTRAN skips the next record.

Examples:

```
READ (1) FIELD1, FIELD2    Read one record from logical unit 1;
                           assign values to variables FIELD1 and
                           FIELD2.

READ (8)                   Advance logical unit 8 one record.
```

11.4.2 Formatted Sequential READ Statement

You use formatted sequential READ statements to transmit information in external format.

The format is:

```
READ (u,f)[[list]]
```

where

```
u    is a logical unit number from 1 to 9
f    is a format statement number
list is an I/O list
```

When the formatted sequential READ statement transfers data from the indicated logical unit, FORTRAN converts transmitted characters to internal format as specified by the format specification. FORTRAN assigns the resulting values to the elements of the I/O list.

If the FORMAT statement associated with a formatted input statement contains a Hollerith constant or alphanumeric literal, input data will be read and stored directly into the format specification. For example, the statements

```
     READ (5,100)
100  FORMAT (5H DATA)
```

cause five characters to be read and stored in the Hollerith format descriptor. If the character string were HELLO, statement 100 would become

```
100  FORMAT (5HHELLO)
```

If there is no H field, the record is skipped.

If the number of elements in the I/O list is less than the number of fields in the input record, the excess portion of the record is discarded. If the number of elements in the list exceeds the number of input fields, an error condition results unless the format specifications state that one or more additional records are to be read (see Section 12.8).

If no I/O list is present, data transfer takes place between the record and the format specification.

Examples:

```
     READ (1,300) ARRAY
300  FORMAT (20F8.2)
```

Read a record from logical unit 1; assign fields to ARRAY.

```
     READ (5,50)
50   FORMAT (25H PAGE HEADING GOES HERE)
```

Read 25 characters from logical unit 5; place them in the FORMAT statement.

9.4.2.1 **CHKEOF Subroutine** - CHKEOF accepts one real, integer, or logical argument. After the next formatted READ operation, this argument will be set to a non-zero value if the logical end-of-file was encountered. Otherwise, it will be set to zero.

Only use CHKEOF when reading one record from the logical unit.

The following is an example of the use of CHKEOF:

```
      .
      .
      .
CALL CHKEOF(EOF)
READ (N,101)DATA
IF (EOF .NE. 0) GO TO 9999
      .
      .
      .
```

11.4.3 Unformatted Direct Access READ Statement

You use an unformatted direct access READ statement to transmit a value or values to a direct access device in internal format.

The format is:

```
READ (u'r) [[list]]
```

where

```
u     is a logical unit number from 1 to 9
r     is the record number
list  is an I/O list
```

The unformatted direct access READ statement positions the input file to a specified record and transfers the fields in that record to the elements in the I/O list without translation.

The logical unit number u may be an unsigned integer constant or a positive integer variable. The record number r may also be a variable. If there are more fields in the input record than elements in the I/O list, FORTRAN discards the excess portion of the record. If there is insufficient data in the record to satisfy the requirements of the I/O list, an error condition results.

The unit number in the unformatted direct access READ statement must refer to a unit that you have previously defined for direct access processing in a DEFINE FILE statement.

Examples:

`READ (1'10) LIST(1),LIST(8)`	Read record 10 of a file on logical unit 1; assign two INTEGER values to specified elements of array LIST.
`READ (4'58) (RHO(N),N=1,5)`	Read record 58 of a file on logical unit 4; assign five real values to array RHO.

11.5 WRITE STATEMENTS

11.5.1 Unformatted Sequential WRITE Statement

You use an unformatted sequential WRITE statement to transmit values in their internal representation to a logical unit.

The format is:

```
WRITE (u)[[list]]
```

where

```
u    is a logical unit number from 1 to 9
list is an I/O list
```

When the unformatted sequential WRITE statement transmits the values of the elements in the I/O list to the specified logical unit, it does so without translation, as one unformatted record.

The logical unit specifier is an integer variable or an integer constant from 1 to 9.

If an unformatted WRITE statement contains no I/O list, one null record is output to the specified unit.

A record may hold 85 single-precision variables. If the list elements fill more than one record, FORTRAN writes successive records until the list is completed. Thus, if there are 100 variables on the list, FORTRAN uses two records; one record contains 85 variables and the second contains 15 variables. For example:

```
DIMENSION X(200)
WRITE (6) X
```

will produce three records on logical unit 6, the first containing X(1) to X(85), the second X(86) to X(170), and the third X(171) to X(200). If the amount of data FORTRAN will transmit exceeds the record size, an error condition results. If the WRITE statement does not completely fill the record with data, FORTRAN zero fills the unused portion of the record.

Examples:

```
WRITE (1) (LIST(K),K=1,5)    Output the contents of elements 1
                             through 5 of array LIST to logical
                             unit 1.

WRITE (4)                    Write a null record on logical unit 4.
```

11.5.2 Formatted Sequential WRITE Statements

You use a formatted sequential WRITE statement to translate a value from its internal representation to character format and then transmit it to a logical unit.

The format is:

```
WRITE (u,f)[[list]]
```

where

```
u    is a logical unit number from 1 to 9
f    is a format statement number
list is an I/O list
```

When the formatted sequential WRITE statement transfers data to the specified logical unit, the I/O list specifies a sequence of values that FORTRAN converts to characters and positions as specified by a format specification.

The logical unit specifier may be an integer variable.

If no I/O list is present, data transfer takes place entirely between the record and the format specification.

The data FORTRAN transmits by a formatted sequential WRITE statement normally constitutes one formatted record. The format specification can, however, specify that additional records are to be written during the execution of that same WRITE statement.

FORTRAN rounds numeric data output under format control during the conversion to external format. (If such data is subsequently input for additional calculations, loss of precision may result. In this case, unformatted output is preferable to formatted output.)

The records FORTRAN transmits by a formatted WRITE statement must not exceed the length that the specified device can accept. For example, a line printer typically cannot print a record that is longer than 132 characters.

Examples:

```
     WRITE (6, 650)                  (Output the contents of the
650  FORMAT (' HELLO, THERE')        FORMAT statement to logical
                                     unit 6.)

     WRITE (1,95) AYE, BEE, CEE      (Write one record of three
95   FORMAT (F8.5, F8.5, F8.5)       fields to logical unit 1.)

     WRITE (1,950) AYE, BEE, CEE     (Write three separate records
950  FORMAT (f8.5)                   of one field each to logical
                                     unit 1.)
```

In the last example, format control arrives at the rightmost parenthesis of the FORMAT statement before all elements of the I/O list have been output. Each time this occurs, FORTRAN terminates the current record and initiates a new record. Thus, FORTRAN writes three separate records (see Section 12.5).

11.5.3 Unformatted Direct Access WRITE Statement

You use an unformatted direct access WRITE statement to transmit a value in its internal representation to a specific record on a direct access device.

The format is:

```
WRITE (u'r) [[list]]
```

where

```
u    is a logical unit number from 1 to 9
r    is the record number
list is an I/O list
```

When the unformatted direct access WRITE statement transmits the values of the elements in the I/O list to a particular record position on a direct access file, the data is written in internal format without translation.

The logical unit specifier may be an integer variable. The record number r may be an unsigned integer constant or integer variable. A record may hold 85 single-precision variables. If the list elements fill more than one record, FORTRAN writes successive records until the list is completed. Thus, if there are 100 variables on the list, FORTRAN uses two records; one record contains 85 variables and the second contains 15 variables. For example

```
DIMENSION X(200)
WRITE (6) X
```

will produce three records on unit 6, the first containing X(1) to X(85), the second X(86) to X(170), and the third X(171) to X(200). If the amount of data FORTRAN will transmit exceeds the record size, an error condition results. If the WRITE statement does not completely fill the record with data, FORTRAN zero fills the unused portion of the record.

Examples:

```
WRITE (2'35) (NUM(K),K=1,10)
```
(Output ten integer values to record 35 of the file connected to logical unit 2.)

```
WRITE (3'J) ARRAY
```
(Output the entire contents of ARRAY to the file connected to logical unit 3 into the record indicated by the value of J.)

11.6 AUXILIARY INPUT/OUTPUT STATEMENTS

You use statements in this category to perform file management functions.

11.6.1 BACKSPACE Statement

Use the BACKSPACE statement to reposition a file to the previous record accessed.

The format is:

```
BACKSPACE u
```

where

u is a logical unit number from 1 to 9

When the BACKSPACE statement repositions a currently open sequential file back one record, it repositions it to the beginning of that record. On the execution of the next I/O statement for that unit, that record is available for processing.

The unit number must refer to a directory structured device (e.g., disk), and a file must be open on that device. If the file is positioned at the first record, FORTRAN ignores the BACKSPACE statement.

Example:

```
BACKSPACE 4     (Reposition open file on logical unit 4 to
                beginning of the previous record.)
```

11.6.2 DEFINE FILE Statement

The DEFINE FILE statement establishes the size and structure of a file upon which FORTRAN will perform direct access I/O.

The format is:

```
DEFINE FILE u (m,n,U,v) [[,u(m,n,U,v)]]...
```

where

- u is an integer constant or variable that specifies the logical unit number
- m is an integer constant or variable that specifies the number of records in the file
- n is an integer constant or variable that specifies the length, in words, of each record
- U specifies that the file is unformatted (binary) and the letter U is the only acceptable entry in this position
- v is an integer variable, called the associated variable of the file

Once you have specified the attributes of a direct access device by means of the DEFINE FILE, you should always specify them in the same manner.

At the conclusion of each direct access I/O operation, FORTRAN assigns the record number of the next higher numbered record in the file to v.

The DEFINE FILE statement specifies that a file containing m fixed-length records of n words each exists, or is to exist, on logical unit u. The records in the file are sequentially numbered from 1 through m.

You must type the DEFINE FILE statement before the first direct access I/O statement that refers to the specified file.

The DEFINE FILE statement also establishes the integer variable v as the associated variable of the file. At the end of each direct access I/O operation, the FORTRAN I/O system places in v the record number of the record immediately following the one just read or written. Because the associated variable always points to the next sequential record in the file (unless you redefine it by an assignment or input statement), you can use direct access I/O statements to perform sequential processing of the file. The logical unit number u cannot be passed as a dummy argument to a DEFINE FILE statement in a subroutine.

In an overlay environment, or when more than one program unit processes the file, place the associated variable in a resident common block.

Example:

```
DEFINE FILE 3 (1000,48,U,NREC)
```

This statement specifies that logical unit 3 is to be connected to a file of 1000 fixed-length records, each record of which is 48 words long. The records are numbered sequentially from 1 through 1000 and are unformatted. After each direct access I/O operation on this file, the integer variable NREC will contain the record number of the record immediately following the one just processed.

11.6.3 **ENDFILE Statement**

The ENDFILE statement writes an end-file record to the specified sequential unit.

The format is:

```
ENDFILE u
```

where

u is a logical unit number from 1 to 9

When you use the ENDFILE statement to write an end-of-file mark on a directory structured device, note that you cannot write additional information to that device after the ENDFILE statement.

You must write the ENDFILE statement to a formatted output file.

No rewind occurs after this statement.

Example:

```
ENDFILE 2   (Output an end-file record to logical unit 2.)
```

11.6.4 REWIND Statement

The REWIND statement repositions a currently open sequential file to be repositioned to the beginning of the file.

The format is:

```
REWIND u
```

where

u is a logical unit number from 1 to 9

Use the REWIND statement to position a directory structured device to its first record.

If the file is at its first record, FORTRAN ignores the REWIND statement.

The unit number in the REWIND statement must refer to a directory structured device (e.g., disk), and a file must be open on that device.

Example:

```
REWIND 3  (Reposition logical unit 3 to beginning of currently
           open file.)
```

CHAPTER 12

FORMAT STATEMENTS

12.1 INTRODUCTION

FORMAT statements are nonexecutable statements used in conjunction with formatted I/O statements. The FORMAT statement describes the format in which FORTRAN transmits data fields, and the data conversion and editing needed to achieve that format.

The FORMAT statement has the form:

```
st FORMAT (glflsl[[f2s2]]...[[fnqn]])
```

where

f	is a field descriptor, or a group of field descriptors enclosed in parentheses
s	is a field separator (either a comma or slash)
q	is zero or more slash (/) record terminators
st	is a mandatory statement number

Including the parentheses is called the format specification. You must enclose the list in parentheses. A field descriptor in a format specification has the form:

```
[[r]]cw[[.d]]
```

where

r	represents a repeat count that specifies that FORTRAN is to apply the field descriptor to r successive fields (If you omit the repeat count, FORTRAN assumes it to be 1.)
c	is a format code
w	is the field width
d	is the number of characters to the right of the decimal point, and should be less than w

The terms r, w, and d must all be unsigned integer constants less than or equal to 255.

The field separators are comma and slash. A slash has the additional function of being a record terminator. The field descriptors used in format specifications are as follows:

- Integer: Iw
- Logical: Lw
- Real, Double-Precision, Complex: Fw.d, Ew.d, Dw.d, Gw.d, Bw.d
- Literal, Editing: Aw, nH, nP, nX, Tn, $, '...', /

(In the alphanumeric and editing field descriptors, n specifies the number of characters or character positions.)

You can precede the F, E, D, or G field descriptors by a scale factor of the form:

nP

where n is an optionally signed integer constant in the range -127 to +127. The scale factor specifies the number of positions the decimal point is to be scaled to the left or right. During data transmission, FORTRAN scans the format specification from left to right. FORTRAN then performs data conversion by correlating the values in the I/O list with the corresponding field descriptors. In the case of H field descriptors and alphanumeric literals, data transmission takes place entirely between the field descriptor and the external record.

12.2 FIELD DESCRIPTORS

The individual field descriptors that can appear in a format specification are described in detail in the following sections. The field descriptors ignore leading spaces in the external field but treat embedded and trailing spaces as zeros.

12.2.1 I Field Descriptor

The I field descriptor governs the translation of integer data.

The format is:

Iw

12.2.1.1 **Input** - The I field descriptor causes an input statement to read w characters from an external record. FORTRAN then assigns the character as an integer value to the corresponding integer element of the I/O list. The external data must be an integer; it must not contain a decimal point or exponent field.

The I field descriptor interprets an all-blank field as a zero value.

If the value of the external field exceeds the range of the corresponding integer list element, an error occurs. If the first non-blank character of the external field is a minus symbol, the I field descriptor causes the field to be stored as a negative value; FORTRAN treats a field preceded by a plus symbol, or an unsigned field, as a positive value.

Examples:

Format	External Field	Internal Representation
I4	2788	2788
I3	-26	-26
I9	312	312
I9	3.12	not permitted; error
I3	-871	-87 (one is lost)

12.2.1.2 **Output** - On output, the I field descriptor transmits the value of the corresponding integer I/O list element, right justified, to an external field w characters in length. It also replaces any leading zeros with spaces. If the value does not fill the field, FORTRAN inserts leading spaces. If the value of the list element is negative, the field will have a minus symbol as its leftmost non-blank character. Space must therefore be included in w for a minus symbol if you expect one to be output. FORTRAN suppresses plus symbols and you need not account for them in w. If w is too small to contain the output value, FORTRAN fills the entire external field with asterisks.

Examples:

Format	Internal Value	External Representation
I3	284	284
I4	-284	-284
I5	174	174
I2	3244	**
I3	-473	***
I7	29.812	not permitted; error

12.2.2 F Field Descriptor

The F field descriptor specifies the data conversion and editing of real or double-precision values, or the real or imaginary parts of complex values.

The format is:

Fw.d

12.2.2.1 **Input** - On input, the F field descriptor causes FORTRAN to read w characters from the external record and to assign the characters as a real value to the corresponding I/O list element. If the first non-blank character of the external field is a minus sign, FORTRAN treats the field as a negative value; FORTRAN assumes a field preceded by a plus sign (or an unsigned field) to be positive. FORTRAN considers an all-blank field to have a value of zero. In all appearances of the F field descriptor, w must be greater than or equal to d+, where the extra character is the decimal point.

If the field contains neither a decimal point nor an exponent, FORTRAN treats it as a real number of w digits, in which the rightmost d digits are to the right of the decimal point. If the field contains an explicit decimal point, the location of that decimal point overrides the location you specify in the field descriptor. If the field contains an exponent, FORTRAN uses the exponent to establish the magnitude of the value before it assigns the value to the list element.

Examples:

Format	External Field	Internal Representation
F8.5	123456789	123.45678
F8.5	-1234.567	-1234.56
F8.5	24.77E+2	2477.0
F5.2	1234567.89	123.45

12.2.2.2 **Output** - On output, the F field descriptor causes FORTRAN to round the value of the corresponding I/O list element to d decimal positions and to transmit an external field w characters in length, right justified. If the converted data consists of fewer than w characters, FORTRAN inserts leading spaces; if the data exceeds w characters, FORTRAN fills the entire field with asterisks.

The field width must be large enough to accommodate: (1) a minus sign, if you expect one to be output (FORTRAN suppresses plus signs); (2) at least one digit to the left of the decimal point; (3) the decimal point itself; and (4) d digits to the right of the decimal. For this reason, w should always be greater than or equal to (d+3).

Examples:

Format	Internal Value	External Representation
F8.5	2.3547188	2.35472
F9.3	8789.7361	8789.736
F2.3	51.44	**
F10.4	-23.24352	-23.2435
F5.2	325.013	*****
F5.2	-.2	-0.20

12.2.3 E Field Descriptor

The E field descriptor specifies the transmission of real or double-precision values in exponential format.

The format is:

Ew.d

12.2.3.1 **Input** - The E field descriptor causes an input statement to input w characters from an external record. It interprets and assigns that data in exactly the same way as the F field descriptor.

Examples:

Format	External Field	Internal Representation
E9.3	734.432E3	734432.0
E12.4	1022.43E-6	1022.43E-6
E15.3	52.37596	52.37596
E12.5	210.5271D+10	210.5271E10

Note that in the last example the E field descriptor ignores the double-precision indicator D and treats it as though it were an E indicator.

12.2.3.2 **Output** - The E field descriptor causes an output statement to transmit the value of the corresponding list element to an external field w characters in width, right justified. If the number of characters in the converted data is less than w, FORTRAN inserts leading spaces; if the number of characters exceeds w, FORTRAN fills the entire field with asterisks. The corresponding I/O list element must be of real, double-precision, or complex type.

FORTRAN transmits data output under control of the E field descriptor in a standard form, consisting of

- a minus sign if the value is negative (plus signs are suppressed)
- a zero
- a decimal point
- d digits to the right of the decimal
- a 3-character exponent of the form:

  E+nnn

  or

  E-nnn

  where nn is a 2-digit integer constant

The d digits to the right of the decimal point represent the entire value, scaled to a decimal fraction.

Because w must be large enough to include a minus sign (if any are expected), a zero, a decimal point, and an exponent, in addition to d digits, w should always be equal to or greater than (d+7).

Examples:

Format	Internal Value	External Representation
E9.2	475867.222	0.48E+06
E12.5	475867.222	0.47587E+06
E12.3	0.00069	0.690E-03
E10.3	-0.5555	-0.556E+00
E5.3	56.12	*****

12.2.4 D Field Descriptor

The D field descriptor specifies the transmission of real or double-precision values.

The format is:

Dw.d

12.2.4.1 **Input** - On input, the D field descriptor functions exactly like an E field descriptor, except that FORTRAN converts the input data and assigns it as a double-precision entity.

Examples:

Format	External Field	Internal Representation
D10.2	12345	12345000.0D0
D10.2	123.45	123.45D0
D15.3	367.4981763D+04	3.674981763D+06

12.2.4.2 **Output** - On output, the effect of the D field descriptor is identical to that of the E field descriptor, except that FORTRAN uses the D exponent field indicator in place of the E indicator.

Examples:

Format	Internal Value	External Representation
D14.3	0.0363	0.363D-01
D23.12	5413.87625793	0.541387625793D+04
D9.6	1.2	*********

12.2.5 B Field Descriptor

The B field descriptor is a convenient method for transmitting double-precision information.

Internally, such a value is identical to a double-precision number. Upon output, the B acts like an F. On input, however, it acts like a D.

12.2.6 G Field Descriptor

The G field descriptor transmits real, double-precision, or complex data in a form that is in effect a combination of the F and E field descriptors.

The format is:

Gw.d

12.2.6.1 **Input** - On input, the G field descriptor functions identically like the F field descriptor.

12.2.6.2 **Output** - On output, the G field descriptor causes FORTRAN to transmit the value of the corresponding I/O list element to an external field w characters in length, right justified. The form in which the value is output is a function of the magnitude of the value, as described in Table 12-1.

Table 12-1
Effect of Data Magnitude on G Format Conversions

Data Magnitude	Effective Conversion
m < 0.1	Ew.d
0.1 < m < 1.0	F(w-4).d, 4X
1.0 < m < 10.0	F(w-4).(d-1), 4X
.	.
.	.
.	.
10d-2 < m < 10d-1	F(w-4).1, 4X
10d-1 < m < 10d	F(w-4).0, 4X
m > 10d	Ew.d

The 4X field descriptor is inserted by the G field descriptor for values within its range; it means that four spaces are to follow the numeric data representation.

The field width, w, must include:

1. space for a minus sign, if any are expected (plus signs are suppressed)
2. at least one digit to the left of the decimal point
3. the decimal point itself
4. d digits to the right of the decimal
5. (for values that are outside the effective range of the G field descriptor) a 4-character exponent

Therefore, w should always be equal to or greater than (d+7).

Examples:

Format	Internal Value	External Representation
G13.6	0.01234567	0.123457E-01
G13.6	-0.12345678	-0.123457
G13.6	1.23456789	1.23457
G13.6	12.34567890	12.3457
G13.6	123.45678901	123.457
G13.6	-1234.56789012	-1234.57
G13.6	12345.67890123	12345.7
G13.6	123456.78901234	123457.
G13.6	-1234567.89012345	-0.123457E+07

For comparison, consider the following example of the same values output under the control of an equivalent F field descriptor.

Format	Internal Value	External Representation
F13.6	0.01234567	0.012346
F13.6	-0.12345678	-0.123457
F13.6	1.23456789	1.234568
F13.6	12.34567890	12.345679
F13.6	123.45678901	123.456789
F13.6	-1234.56789012	-1234.567890
F13.6	12345.67890123	12345.678901
F13.6	123456.78901234	123456.789012
F13.6	-1234567.89012345	**********

12.2.7 **L Field Descriptor**

The L field descriptor specifies the transmission of logical data.

The format is:

Lw

12.2.7.1 **Input** - The L field descriptor causes an input statement to read w characters from external record. If the first non-blank character of that field is the letter T or the string .T, FORTRAN assigns the value .TRUE. to the corresponding I/O list element. (The corresponding I/O list element must be of logical type.) If the first non-blank character of the field is the letter F or the string .F, or if the entire field is blank, FORTRAN assigns the value .FALSE. . Any other value in the external field causes an error condition.

12.2.7.2 **Output** - The L field descriptor causes an output statement to transmit either the letter T, if the value of the corresponding list element is .TRUE. or the letter F, if the value is .FALSE., to an external field w characters wide. The letter T or F is in the rightmost position of the field, preceded by (w-1) spaces.

Examples:

Format	Internal Value	External Representation
L5	.TRUE.	T
L1	.FALSE.	F

12.2.8 A Field Descriptor

The A field descriptor specifies the transmission of alphanumeric data.

The format is:

Aw

12.2.8.1 **Input** - On input, the A field descriptor causes w characters to be read from the external record and stored in ASCII format in the corresponding I/O list element. (The corresponding I/O list element may be of any data type.) The maximum number of characters that FORTRAN can store in a variable or array element depends on the data type of that element, as listed in Table 12-2.

Table 12-2
Character Storage

I/O List Element	Maximum Number of Characters
Logical	6
Integer	6
Real	6
Double-Precision	12
Complex	12

If w is greater than the maximum number of characters that FORTRAN can store in the corresponding I/O list element, only the rightmost six or twelve characters (depending on the data type of the variable or array element) are assigned to that entity; the leftmost excess characters are lost. If w is less than the number of characters that FORTRAN can store, it assigns w characters to the list element, left justified, and adds trailing spaces to fill the variable or array element.

Examples:

Format	External Field	Internal Representation
A6	PAGE #	PAGE # (Integer)
A6	PAGE #	GE # (Real)
A12	PAGE #	PAGE # (Double Precision)

12.2.8.2 **Output** - On output, the A field descriptor causes FORTRAN to transmit the contents of the corresponding I/O list element to an external field w characters wide. If the list element contains fewer than w characters, the data appears in the field right justified with leading spaces. If the list element contains more than w characters, FORTRAN transmits only the leftmost w characters.

Examples:

Format	Internal Value	External Representation
A5	OHMS	OHMS
A5	VOLTS	VOLTS
A5	AMPERES	AMPER

12.2.9 H Field Descriptor

The format is:

nHccc...c

where

n specifies the number of characters to be transmitted
c is an ASCII character

When the H field descriptor appears in a format specification, data transmission takes place between the external record and the field descriptor itself.

The H field descriptor causes an input statement to read n characters from the external record and to place them in the field descriptor, with the first character appearing immediately after the letter H. FORTRAN replaces any characters that had been in the field descriptor prior to input by the input characters.

The H field descriptor causes an output statement to transmit the n characters in the field descriptor following the letter H to the external record. An example of the use of H field descriptors for input and output follows:

```
      WRITE (4,100)
100   FORMAT (41H ENTER PROGRAM TITLE, UP TO 20 CHARACTERS)
      READ (4,200)
200   FORMAT (20H  TITLE GOES HERE   )
```

The WRITE statement transmits the characters from the H field descriptor in statement 100 to the user's terminal. The READ statement accepts the response from the keyboard, placing the input data in the H field descriptor in statement 200. The new characters replace the string TITLE GOES HERE; if you enter fewer than 20 characters, FORTRAN fills the remainder of the H field descriptor with spaces to the right.

12.2.9.1 **Alphanumeric Literals** - In an output statement, you may use an alphanumeric literal in place of an H field descriptor; both types of format specifiers function identically. However, you cannot use an alphanumeric literal on input.

You write an apostrophe character within an alphanumeric literal as two apostrophes. For example:

```
50 FORMAT (' TODAY''S DATE IS: ',I2,'/',I2,'/',I2)
```

FORTRAN treats a pair of apostrophes used in this manner as a single character.

12.2.10 **X Field Descriptor**

The X field descriptor causes spaces to be skipped in a record.

The format is:

```
nX
```

When used in an input statement, the spaces skipped as a result of the x field descriptor are represented by the next n characters in the input record.

In an output statement, the X field descriptor causes n spaces to be transmitted to the external record. For example:

```
      WRITE (5,90) NPAGE
90    FORMAT (13H1PAGE NUMBER ,I2,16X,23HGRAPHIC ANALYSIS, CONT.)
```

The WRITE statement prints a record similar to:

```
PAGE NUMBER nn           GRAPHIC ANALYSIS, CONT.
```

where "nn" is the current value of the variable NPAGE. FORTRAN does not print the numeral 1 in the first H field descriptor, but instead uses it to advance the printer paper to the top of a new page. Printer carriage control is explained in Section 12.6.

12.2.11 T Field Descriptor

The T field descriptor is a tabulation specifier.

The format is:

```
Tn
```

where

n indicates the character position of the external record. The value of n must be greater than or equal to one, but not greater than the number of characters allowed in the external record.

12.2.11.1 **Input** - On input, the T field descriptor causes FORTRAN to position the external record to its nth character position. For example, if a READ statement inputs a record containing

```
ABC   XYZ
```

under control of the FORMAT statement

```
10   FORMAT (T7,A3,T1,A3)
```

the READ statement would input the characters XYZ first, then the characters ABC.

12.2.11.2 **Output** - On output to devices other than the line printer or terminal, the T field descriptor states that subsequent data transfer is to begin at the nth character position of the external record. (For output to a printing device, data transfer begins at position n-1). This is because FORTRAN reserves the first position of a printed record for a carriage control character (see Section 12.6), which is never printed.

Thus, the statements

```
     WRITE(4,25)
25   FORMAT (T51,'COLUMN 2',T21,'COLUMN 1')
```

would cause the following line to be printed:

```
              Position 20                Position 50

                   COLUMN 1                   COLUMN 2
```

12.2.12 $ Descriptor

The dollar sign character ($) appearing in a format specification modifies the carriage control specified by the first character of the record. The $ descriptor is intended primarily for interactive I/O and causes the terminal print position to be left at the end of the written text (rather than returned to the left margin) so that a typed response will appear on the same line following the output.

Example:

```
          A=5
          WRITE (4,100) A
          READ (4,200) B
100       FORMAT (' SAMPLE NO.', I2, ' IS: ',$)
200       FORMAT (A6)
          WRITE (4,200) B
          END
```

This program outputs

```
SAMPLE NO.  5 IS:  RED
RED
```

12.3 COMPLEX DATA EDITING

Since a complex value is an ordered pair of real values, input or output of a complex entity is governed by two real field descriptors, using any combination of the forms Fw.d, Ew.d, Dw.d, or Gw.d.

12.3.1 Input

On input, FORTRAN reads two successive fields and assigns them to a complex I/O list element as its real and imaginary parts, respectively.

Examples:

Format	External Fields	Internal Representation
F8.5,f8.5	1234567812345.67	123.45678, 12345.67
E9.1,f9.3	734.432E8123456789	734.432E8, 12345.678

12.3.2 Output

On output, FORTRAN transmits the constituent parts of a complex value under the control of repeated or successive field descriptors. Nothing intervenes between those parts unless explicitly stated by the format specification.

Examples:

Format	Internal Values	External Representation
2F8.5	2.3547188, 3.456732	2.35472 3.45673
E9.2,' , ',E5.3	47587.222, 56.123	0.48E+06 , *****

12.4 SCALE FACTOR

Through the use of a scale factor, you can alter the location of the decimal point in real, double-precision, and complex values during input or output.

The format is:

nP

where

n is a signed or unsigned integer constant in the range -127 to +127 specifying the number of positions the decimal point is to be moved to the right or left.

You may place a scale factor anywhere in a format specification, but it must precede the field descriptors with which it is to be associated. It has the forms:

nPFw.d nPEw.d nPDw.d nPGw.d

Data input under control of one of the above field descriptors is multiplied by 10**-n before FORTRAN assigns it to the corresponding I/O list element. For example, a 2P scale factor multiplies an input value by .01, moving the decimal point two places to the left; a -2P scale factor multiplies an input value by 100, moving the decimal point two places to the right. If the external field contains an explicit exponent, however, the scale factor has no effect.

Examples:

Format	External Field	Internal Representation
3PE10.5	37.614	.037614
3PE10.5	37.614E2	3761.4
-3PE10.5	37.614	37614.

The effect of the scale factor on output depends on the type of field descriptor with which it is associated. For the F field descriptor, FORTRAN multiplies the value of the I/O list element by 10**N before it transmits it to the external record. Thus, a positive scale factor moves the decimal point to the right; a negative scale factor moves the decimal point to the left.

FORTRAN adjusts values output under control of an E or D field descriptor with a scale factor by multiplying the basic real constant portion of each value by 10**N and subtracting n from the exponent. Thus a positive scale factor moves the decimal point to the right and decreases the exponent; a negative scale factor moves the decimal point to the left and increases the exponent.

FORTRAN suspends the effect of the scale factor while the magnitude of the data to be output is within the effective range of the G field descriptor, since G supplies its own scaling function. The G field descriptor functions as an E field descriptor when the magnitude of the data value is outside its range; the effect of the scale factor is therefore the same as described for that field descriptor.

Note that on input, and on output under control of an F field descriptor, a scale factor actually alters the magnitude of the data; otherwise, a scale factor attached to an E, D, or G field descriptor merely alters the form in which the data is transmitted. Note also that on input a positive scale factor moves the decimal point to the left and a negative scale factor moves the decimal point to the right, while on output the effect is just the reverse.

If you do not attach a scale factor to a field descriptor, FORTRAN assumes a scale factor of zero. Once you specify a scale factor, however, it applies to all subsequent real and double-precision field

descriptors in the same format specification, unless another scale factor appears. You may only reinstate a scale factor of zero by an explicit 0P specification.

Some examples of scale factor effect on output are:

Format	Internal Value	External Representation
1PE12.3	-270.139	-2.701E+02
1PE12.2	-270.139	-2.70E+02
-1PE12.2	-270.139	-0.03E+04

12.5 GROUPING AND GROUP REPEAT SPECIFICATIONS

You can apply any field descriptor (except H, T, P, or X) to a number of successive data fields by preceding that field descriptor with an unsigned integer constant, called a repeat count, that specifies the number of repetitions. For example, the statements

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

and

```
20   FORMAT (3E12.4,4I5)
```

have the same effect.

Similarly, you may repeatedly apply a group of field descriptors to data fields by enclosing those field descriptors in parentheses, with an unsigned integer constant, called a group repeat count, preceding the left parenthesis. For example:

```
50   FORMAT (2I8,3(F8.3,E15.7))
```

is equivalent to:

```
50   FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7)
```

You can enclose an H or X field descriptor, which could not otherwise be repeated, in parentheses. FORTRAN then treats it as a group repeat specification, thus allowing it to be repeated a desired number of times.

If you omit a group repeat count, FORTRAN assumes it to be 1.

12.6 CARRIAGE CONTROL

FORTRAN never transmits the first character of a record to a printing device; instead, FORTRAN interprets this first character as a carriage control character. The FORTRAN I/O system recognizes certain characters for this purpose; these characters and their effects are shown in Table 12-3.

Table 12-3
Carriage Control Characters

Character	Effect
space	Advance one line
0 (zero)	Advance two lines
1 (one)	Advance to top of next page
+ (plus)	Do not advance (allows overprinting)

FORTRAN treats any character other than those described in Table 12-3 as though it is a space, and deletes it from the print line.

12.7 FORMAT SPECIFICATION SEPARATORS

In a format specification you generally separate field descriptors from one another by commas. You may also use the slash (/) record terminator to separate field descriptors. A slash causes FORTRAN to terminate the input or output of the current record and to initiate a new record. You may omit the comma when using a slash. Also, you need not type a comma after a Hollerith constant.

Example:

```
      WRITE (5,40) K,L,M,N,O,P
40    FORMAT (3A6/I6,2F8.4)
```

is equivalent to

```
      WRITE (5,40) K,L,M
40    FORMAT (3A6)
      WRITE (5,50) N,O,P
50    FORMAT (I6,2F8.4)
```

It is possible to bypass input records or to output blank records by the use of multiple slashes. If n consecutive slashes appear between two field descriptors, they cause FORTRAN to skip (n-1) records on input or (n-1) blank records to be output. (The first slash terminates the current record; the second slash terminates the first skipped or blank record, and so on.) If n slashes appear at the beginning or end of a format specification, however, they result in n skipped or blank records, because the initial and terminal parentheses of the format specification are themselves a record initiator and record terminator, respectively. An example of the use of multiple record terminators is:

```
      WRITE (5,99)
99    FORMAT ('1'T51'HEADING LINE'//T51'SUBHEADING LINE'//)
```

The above statements output the following:

```
                    Column 50, top of page

                              HEADING LINE
                    (blank line)
                              SUBHEADING LINE
                    (blank line)
                    (blank line)
```

12.7.1 External Field Separators

A field descriptor such as fw.d specifies that an input statement is to read w characters from the external record. If the data field in question contains fewer than w characters, the input statement would read some characters from the following field. To avoid this, you can pad the short field with leading zeros or spaces. Padding is unnecessary, however, if you terminate an input field containing fewer than w characters by a comma. The comma overrides the field descriptor's field width specification. This practice, called short field termination, is particularly useful when entering data from a terminal keyboard. You may also use it in conjunction with I, F, E, D, G, and L field descriptors.

Examples:

```
      READ (6,100) I,J,A,B
100   FORMAT (2I6,2F10.2)
```

If the external record input by the above statements contains

```
1,-2,1.0,35
```

then the following assignments take place:

```
I = 1

J = -2

A = 1.0

B = 0.35
```

Note that the physical end of the record also serves as a field terminator. Note also that the d part of a w.d specification is not affected, as illustrated by the assignment to B.

You may only terminate fields of fewer than w characters by a comma. If you use a comma after a field of w characters or greater, FORTRAN will consider the comma to be part of the following field.

Two successive commas, or a comma following a field of exactly w characters, constitutes a null (zero-length) field. Depending on the field descriptor in question, the resulting value assigned is 0, 0.0, 0D0, or .FALSE..

You cannot use a comma to terminate a field that is to be read under control of an A, H, or alphanumeric literal field descriptor. If FORTRAN encounters the physical end of the record before it has read w characters, however, short field termination is accomplished and FORTRAN can assign the characters that were input. It also appends trailing spaces to fill the corresponding I/O list element or the field descriptor.

12.8 FORMAT CONTROL INTERACTION WITH INPUT/OUTPUT LISTS

FORTRAN initiates format control with the beginning of execution of a formatted I/O statement. The action of format control depends on information provided jointly by the next element of the I/O list (if one exists) and the next field descriptor of the FORMAT statement. FORTRAN interprets both the I/O list and the format specification from left to right.

If the I/O statement contains an I/O list, at least one field descriptor of a type other than H, X, T, or P must exist in the format specification. Otherwise, an execution error occurs.

When FORTRAN executes a formatted input statement, it reads one record from the specified unit and initiates format control; thereafter, additional records can be read as indicated by the format specification. Format control demands that a new record be input whenever a slash is encountered in the format specification, or when the rightmost parenthesis of the format specification is reached and additional I/O list elements remain.

Each field descriptor of types I, F, E, D, G, L, and A corresponds to one element in the I/O list. No list element corresponds to an H, X, P, T, or alphanumeric literal field descriptor. In the case of H and alphanumeric literal field descriptors, data transfer takes place directly between the external record and the format specification.

When format control encounters an I, F, E, D, G, L, or A field descriptor, it determines if a corresponding element exists in the I/O list. If so, format control transmits data, appropriately converted to or from external format, between the record and the list element, then proceeds to the next field descriptor (unless the current one is to be repeated). If there is no corresponding list element, format control terminates.

When FORTRAN reaches the rightmost parenthesis of the format specification, it determines whether or not there are more I/O list elements to be processed. If not, format control terminates. If additional list elements remain, however, FORTRAN terminates the current record and initiates a new one. Format control then reverts to the rightmost, top-level group repeat specification (the one whose left parenthesis matches the next-to-last right parenthesis of the format specification). If no group repeat specification exists in the format specification, format control returns to the initial left parenthesis of the format specification. Format control then continues from that point.

12.9 SUMMARY OF RULES FOR FORMAT STATEMENTS

The following is a summary of the rules pertaining to the construction and use of the FORMAT statement and its components, and to the construction of the external fields and records with which a format specification communicates.

12.9.1 General

- You must always label a FORMAT statement.
- In a field descriptor such as rIw or nX, the terms r, w, and n must be unsigned integer constants greater than zero. You may omit the repeat count and field width specification.
- In a field descriptor such as Fw.d, the term d must be an unsigned integer constant. It must be present in F, E, D, and G field descriptors even if it is zero. The decimal point must also be present. The field width specification w must be greater than d. The specifications w and d must occur together or not at all.

- In a field descriptor such as nHcc...c, exactly n characters must be present following the H format code. Any ASCII character may appear in this field descriptor (an alphanumeric literal field descriptor follows the same rule).

- In a scale factor of the form nP, n must be a signed or unsigned integer constant in the range -127 to +127 inclusive. Use of the scale factor applies to F, E, D, and G field descriptors only. Once you specify a scale factor, it applies to all subsequent real or double-precision field descriptors in that format specification until another scale factor appears; FORTRAN requires an explicit 0P specification to reinstate a scale factor of zero.

- FORTRAN does not permit a repeat count in H, X, T, or alphanumeric literal descriptors unless you enclose those field descriptors in parentheses and treat them as a group repeat specification.

- If an I/O list is present in the associated I/O statement, the format specification must contain at least one field descriptor of a type other than H, X, P, T, or alphanumeric literal.

12.9.2 Input

- You must precede an external input field with a negative value by a minus symbol; you may optionally precede a positive value by a plus sign.

- An external field whose input conversion is governed by an I field descriptor must have the form of an integer constant; it cannot contain a decimal point or an exponent.

- An external field whose input conversion is governed by an F, E, or G field descriptor must have the form of an integer constant or a real or double-precision constant; it can contain a decimal point and/or an E or D exponent field.

- If an external field contains a decimal point, the actual size of the fractional part of the field, as indicated by that decimal point, overrides the d specification of the corresponding real or double-precision field descriptor.

- If an external field contains an exponent, it causes the scale factor (if any) of the corresponding field descriptor to be inoperative for the conversion of that field.

- The field width specification must be large enough to accommodate, in addition to the numeric character string of the external field, any other characters that can be present (algebraic sign, decimal point, and/or exponent).

- A comma is the only character that is acceptable for use as an external field separator. You use it to terminate input of fields that are shorter than the number of characters expected, or to designate null (zero-length) fields.

12.9.3 Output

- A format specification must not demand the output of more characters than can be contained in the external record. (For example, a line printer record cannot contain more than 133 characters, including the carriage control character.)

- The field width specification w must be large enough to accommodate all the characters that FORTRAN may generate by the output conversion, including an algebraic sign, decimal point, and exponent. (The field width specification in an E field descriptor, for example, should be large enough to contain (d+7) characters.)

- FORTRAN uses the first character of a record output to a line printer or terminal for carriage control; FORTRAN never prints it. The first character of such a record should be a space, 0,1,$, or +. FORTRAN treats any other character as a space and deletes it from the record.

CHAPTER 13

FORTRAN IV LIBRARY

The OS/8 FORTRAN IV system contains a general purpose FORTRAN library FORLIB.RL, which may be extended and modified by the librarian LIBRA. The library allows you to compute arithmetic and transcendental functions, use the complex and double-precision options of the FPP, read console switches, and interface with standard laboratory peripherals.

You use the OS/8 FORTRAN librarian, LIBRA, to create and maintain libraries of RALF modules. The loader uses one such library, specified by the user, to resolve undefined external symbols. Each library contains a collection of RALF modules and a catalog, which lists the program section names and entry points defined in the modules, along with sufficient information for the loader to find them.

LIBRA's tasks are: to create libraries (and their catalogs) from user-specified sets of modules (RALF output files); to add new modules to existing libraries; to copy the contents of a library to a new library (with various options on selective deletion and replacement during the copy); and to list the catalogs of libraries.

To call LIBRA, type

```
.R LIBRA
```

in response to the dot generated by the Keyboard Monitor. LIBRA loads the OS/8 Command Decoder, which prints an asterisk at the left margin. In response to the Command Decoder's asterisk, type in the following order:

1. The output device and name of the library to be created (LIBRA assigns the extension .RL unless one is specified). If no output file is specified, the default name FORLIB.RL is used and output is to the system device.

2. The desired number of index blocks (decimal, maximum 255) enclosed in square brackets. LIBRA allocates two index blocks if no specification is given.

3. The output device for the catalog listing when the library build is complete (preceded by a comma). If no device is specified, the listing is suppressed.

4. The input files (RALF output modules) or libraries to be included in the library (preceded by a backarrow or left angle bracket).

5. Options:

   /C to continue input specification on next line
   /I to make a decision on insertion of each entry point or section name
   /Z to replace an existing file of the same name by the new library
   /R to replace a module of the same name already in the library by a new input file
   = to allow extra blocks for library expansion

The following lines may now be on the terminal:

```
.R LIBRA
*LIB1.RL[5],TTY:<LIB0.RL,R1,R2,R3,R4,R5,R6/Z=20
```

With the above command, you create a library named LIB1.RL on the system device containing the existing library, LIB0.RL, and the files R1, R2,..., R6. You allocate five blocks for the index; cause the catalog to be printed on the console terminal and 20 (octal) extra blocks reserved for future expansion. The /Z indicates that if a file already exists with the name LIB1.RL, the newly created library will replace it.

If you wish to include more than nine modules, type /C to continue input specification on the next line. Note that you must specify the "=" option and the output device for the catalog listing on the last line (that is, the one without /C). The /Z, if it is used, must appear on the first line. Thus:

```
.R LIBRA
*LIB1.RL/Z[5]<R1,R2,R3,R4,R5,R6,R7,R8,R9,R10/C
*,TTY:<R10,R11=20
```

The library now contains the additional files R7, R8,..., R11. You can specify the /I and /R options at any point in the command line; both /I and /R apply only to modules listed on the line in which they appear.

To expand a previously created library, call LIBRA as usual. Specify the name of the old library file as the first output file, the catalog listing file, if desired, and then the modules or libraries to be added as input. Do not specify /Z. Thus:

```
.R LIBRA
*LIB1.RL,TTY:<ROUT,MOD
```

LIBRA adds the contents of ROUT and MOD to LIB1. If the old library file name does not exist, a new library is created using default options if necessary. Since LIBRA cannot change the size of the index or the room left for expansion at this time, it is useless to specify index blocks and expansion blocks.

If by adding a module entry point or section name to a library you duplicate a name in the library catalog, the duplicate name is printed on the terminal. The name in the catalog continues to refer to the original module, unless:

1. You specify /R on a command line. The new module then becomes a library file and the old module of the same name is deleted (unless there are other names for the old module, in which case only the duplicate name is deleted). For example:

   ```
   *LIB1,RL<LIB0,RL,R1,R2,R3/R
   ```

   causes any of the input modules R1, R2, and R3 to replace existing modules in LIB0.RL with the same entry point or section name.

2. You specify /I on the LIBRA command line. Input file entry points and section names are then listed on the console terminal. If the names duplicate names in the catalog, the message printed is:

   ```
   xxxx IS DUPLICATE NAME; KEEP OLD OR NEW?
   ```

   where xxxx is an entry point or section name. You then type OLD and a RETURN (or just a RETURN or O and a RETURN) to keep the old name; NEW and a RETURN (or N and a RETURN) to delete the old name and include the new. The question is repeated if you type any other character.

   If the new names do not appear in the catalog, the message typed is:

   ```
   xxxx:  INCLUDE?
   ```

   where xxxx is the new entry point or section name.

   Type YES and a RETURN (or just a RETURN or Y and a RETURN) to include the name; NO and a RETURN (or N and a RETURN) to omit it. The question is repeated if you type any other character.

You can obtain a catalog listing at any time by omitting the input file specification in the call to LIBRA. For example:

```
.R LIBRA
*FORLIB,RL,LPT:<
```

prints the catalog of FORLIB on the line printer. LIBRA's version number (Vxx) is output as part of the catalog heading.

Entry points and section names may be deleted from the catalog by combining the /I and /Z options. Each catalog entry is listed on the console terminal with the message:

```
name: INCLUDE?
```

Type Y and RETURN to include the section name or entry point; type N and RETURN to delete it. If all catalog entries corresponding to a particular module are deleted from the catalog in this manner, the module is deleted from the library and the message:

```
MODULE IS DELETED
```

is printed on the console terminal.

FORLIB.RL, the standard library supplied with the FORTRAN IV system, contains functions and subroutines that perform mathematical calculations and drive various peripheral devices. You may modify this library with LIBRA to fit the needs of your installation. Although at least one copy of the standard library should be maintained as a backup, it may be desirable to delete unwanted routines from FORLIB in order to reduce storage requirements. For example, you may delete double-precision routines if your installation does not include an FPP-12 with extended precision option. Take care not to delete subroutines that may be called by the various system programs or by other library routines that are not deleted. Table 13-1 lists the library routines that execute calls to entry points in other routines; in general, when an entry in the right column of Table 13-1 is deleted, the corresponding entry in the left column may not be called.

Table 13-1
FORLIB Calling Relationships

Section Name	Entry Point Called
SYNC	DISP, ONQI
DISP	ONQB
EXPIR	EXP3
EXP3	ALOG, EXP
ALOG10	ALOG
COS	SIN
TAN	SIN, COS
SIND	SIN
COSD	SIN
TAND	TAN
ASIN	ATAN, SQRT
ACOS	ATAN, SQRT
ATAN2	ATAN
SINH	EXP
COSH	EXP
TANH	SINH, COSH

For example, to delete the entry points ABS, IABS, and LSW from the catalog, the proper command to LIBRA is:

```
.R LIBRA
*LIB2,RL<LIB1,RL/I/Z
```

Respond with Y and a carriage return to all of the messages except:

```
IABS: INCLUDE? N
ABS: INCLUDE? N
MODULE IS DELETED

.
LSW: INCLUDE? N
.
.
```

The module containing ABS and IABS is deleted from the library because all of its section names and entry points have been deleted from the catalog. Entry point LSW is deleted from the catalog, but the corresponding module remains in the library because other entry points are still present in the catalog. Table 13-2 lists the FORLIB entry points that are contained in modules with different section names.

Table 13-2
FORLIB Multiple Entry Points by Section

Section Name	Entry Points
IABS	ABS
SIGN	ISIGN
AMIN0	AMIN1, MIN0, MIN1
AMAX0	AMAX1, MAX0, MAX1
DIM	IDIM
PLOT	SCALE, CLRPLT, #DISP
REALTM	SAMPLE, ADB
CHARS	CGET, CPUT, CHAR
IFIX	AINT, INT
AMOD	MOD
RSW	LSW, SSW, ROPEN, EXTLVL, RCLOSE
ONQI	ONQB
SYNC	CLOCK, TIME, #CLINT

The catalog entries #FIX, #RFDV, #LTR, #EQ, #NE, #GE, #LE, #GT, #LT, #EXPIR, #CLINT, and #EXPII are used by the compiler and should not be deleted.

13.1 LIBRARY FUNCTIONS AND SUBROUTINES

Library functions and subroutines are called in the same manner as user-written functions and subroutines. The following section lists the library components that are available to FORTRAN programs and illustrates some calling sequences. Arguments must be of the correct number and type but need not have the same name as those shown in the examples. Routines that require LAB8/E or PDP-12 hardware are marked with an asterisk. Routines that will run on the FPP with extended-precision option are marked with two asterisks. You must not use either symbol in the actual FORTRAN program. Certain library routines are used by the FORTRAN system programs and are not available to a user's FORTRAN program. You can identify these routines by the initial "#" character in the entry point or section name; they are not listed in the following section.

13.1.1 ABS (Single-Precision Absolute Value)

ABS calculates the absolute value of a real variable by leaving the variable unchanged if it is positive (or zero) and negating the variable if it is negative.

13.1.2 ACOS (Single-Precision Arc-Cosine Function)

ACOS calculates and returns the primary arc-cosine (in radians) of a real argument less than or equal to 1.0 according to the relation:

If x > 0.0, $ACOS(x) = ATAN\left[\frac{SQRT(1-X\hat{}2)}{x}\right]$

If x < 0.0, $ACOS(x) = \pi + ATAN\left[\frac{SQRT(1-X\hat{}2)}{x}\right]$

If x = 0.0, $ACOS(x) = \pi/2.0$

13.1.3 ADB* (Return Next Sample from Real-Time Sampling Buffer)

ADB finds and returns the next sample in the range [-512, 511] from the real-time sampling buffer. The following program illustrates how ADB may be used to sample 500 points from channel 3 and plot them on the scope:

```
          DIMENSION PLTBUF(400),DATBUF(50)
1         CALL CLRPLT(400,PLTBUF)
          CALL REALTM (DATBUF,50,3,1,500)
          CALL CLOCK (8,10)
          DO 100 I=1,500
100       CALL PLOT(1,I/384.,ADB(X)/1024.+.5)
          READ(1,10)Q
10        FORMAT(I2)
          GO TO 1
          STOP
          END
```

After finishing the plotting, the program waits for you to type the RETURN key, and then repeats the sampling-display process. Note that REALTM sets up the sampling procedure, while CLOCK actually initiates the sampling.

13.1.4 ADC* (Asynchronous Sampling)

The ADC function accepts an integer argument in the range [0,15], assumed to be a channel number. It returns the current value of the referenced channel as a real number in the range [-1, 1]. Sampling employs the fast SAM instruction for one or multiple channels. ADC may not be used in a program that also uses REALTM. The following program illustrates the use of the ADC function.

```
C   EXAMPLE OF ADC FUNCTION
C   REQUIRES PDP12 OR LAB8E HARDWARE
C   SAMPLES AND TYPES ANALOG INPUT
C
   10 CONTINUE
      WRITE(4, 100)
  100 FORMAT('   TYPE IN CHANNEL NUMBER '
     1   'AND NUMBER OF SAMPLES')
      READ(4,101) NC,NS
  101 FORMAT(2I3)
      DO 20 I=1,NS
      X=ADC(NC)
      WRITE(4,102) X
  102 FORMAT(F15.5)
   20 CONTINUE
      GOTO 10
      CALL EXIT
      END
```

13.1.5 AIMAG** (Complex-to-Imaginary Conversion)

AIMAG returns the imaginary part of its complex argument as a real variable.

13.1.6 AINT (Single Precision-Floating Point to Integer

AINT is a floating-point truncation function. Given a real argument, it truncates the fractional part of the argument and returns the integral part as an integer. This is accomplished by taking the absolute value of the argument, aligning and normalizing this result, then restoring the original sign. AINT, IFIX, and INT perform identical functions.

13.1.7 ALOG (Single-Precision Natural Logarithm)

ALOG calculates and returns the natural (Naperian) logarithm of a real argument greater than zero. Any negative or zero argument returns an error message and a value of 0.0. The algorithm used is an eight-term Taylor series approximation.

13.1.8 ALOG10 (Single-Precision Common Logarithm)

ALOG10 calculates and returns the common (base 10) logarithm of a real argument greater than zero. Any negative or zero argument returns an error message and a value of 0.0. The calculation is accomplished by calling ALOG to compute the natural logarithm and executing a change of base.

13.1.9 AMAX0 (Single-Precision Maximum Value)

AMAX0 accepts an arbitrary number of integer arguments and returns a real value equal to the largest of the arguments.

13.1.10 AMAX1 (Single-Precision Maximum Value)

AMAX1 accepts an arbitrary number of real arguments and returns a real value equal to the largest of the arguments.

13.1.11 AMIN0 (Single-Precision Minimum Value)

AMIN0 accepts an arbitrary number of integer arguments and returns a real value equal to the smallest of the arguments.

13.1.12 AMIN1 (Single-Precision Minimum Value)

AMIN1 accepts an arbitrary number of real arguments and returns a real value equal to the smallest of the arguments.

13.1.13 AMOD (Single-Precision A Modulo B)

AMOD accepts two real arguments and returns a real value equal to the remainder when the first argument is divided by the second argument. If the second argument is not sufficiently large to prevent overflow, an error message and a value of 0.0 are returned.

13.1.14 ASIN (Single-Precision Arc-Sine)

ASIN calculates and returns the arc-sine (in radians) of a real argument in the range [-1, 1] according to the relation:

```
ASIN(X) = ATAN(X/SQRT(1-X**2))
```

If the argument falls outside the range [-1, 1], an error message results.

13.1.15 ATAN (Single-Precision Arc-Tangent)

ATAN calculates and returns the primary arc-tangent (in radians) of a real argument. The argument is first reduced according to the relations:

```
(1) If x<2^-14,     atan(x) = x
(2) If x>2^-14,     atan(x) = 1/x
(3) If x>1.0,       atan(x) = /2 - atan(1/x)
(4) If x<0,         atan(x) = -atan(-x)
```

The arc-tangent is then computed by a power series approximation.

13.1.16 ATAN2 (Single-Precision Arc-Tangent of Two Arguments)

ATAN2 accepts two real arguments, one of which is assumed to be an abscissa and the other an ordinate. It calculates the arc-tangent of the quotient of the first argument divided by the second argument. This is accomplished by calling ATAN to find the principal arc-tangent of the quotient and then adjusting the result, depending upon the quadrant in which a point defined by the arguments falls, according to the relations:

```
argument in first quadrant          atan2(y,x) = atan(y/x)
argument in second quadrant         atan2(y,x) = atan(y/x)-
argument in third quadrant          atan2(y,x) = atan(y/x)-
argument in fourth quadrant         atan2(y,x) = atan(y/x)+
```

13.1.17 CABS** (Complex Absolute Value)

CABS accepts a complex argument and returns the absolute value of the argument as a real variable defined by:

```
CABS(X+iY) = SQRT(X**2+Y**2)
```

13.1.18 CCOS** (Complex Cosine)

CCOS accepts a complex argument and returns the cosine of the argument, a complex number defined by:

```
CCOS(X+iY) = COS(X)*COSH(Y)-i*SIN(X)*SINH(Y)
```

13.1.19 CEXP** (Complex Exponential)

CEXP accepts a complex argument and returns the exponential function of the argument, a complex variable defined by:

```
CEXP(X+iY) = EXP(X)*(COS(Y)+i*SIN(Y))
```

13.1.20 CGET (Character Get Subroutine)

The calling sequence:

```
CALL CGET (STRING,N,CHAR)
```

causes the Nth character to be unpacked from STRING and stored in CHAR as a variable in the range 0, 63, where STRING is a character string in A6 format.

13.1.21 CHKEOF (Check for End-of-File Subroutine)

CHKEOF accepts one real, integer, or logical argument. After the next formatted read operation, this argument will be set to non-zero if the logical end-of-file was encountered, or to 0 if the logical end-of-file was not encountered. The following is an example of the use of CHKEOF:

```
      .
      .
      .
      CALL CHKEOF(EOF)
      READ (N,101)DATA
      IF (EOF.NE.0) GO TO 999
      .
      .
      .
```

13.1.22 CLOCK* (Initialize Clock Subroutine)

The purpose of the CLOCK subroutine is to initialize certain features of the KW12A or DK8ES real-time clock. The calling sequence is:

```
CALL CLOCK (FUNCTN,RATE)
```

Depending upon the arguments FUNCTN and RATE, CLOCK can enable Schmitt triggers and clock-controlled A/D conversions, or run the clock at a variable rate. The clock is always run on interrupt. Both arguments may be either integer, real, or logical in type. The first argument inidicates a class of clock functions, and the second specifies a clock rate in Hertz. A common use of the clock routine occurs in conjunction with the REALTM subroutine. With one exception noted

below, the clock routine is independent of hardware type. That is, a program employing the KW12A clock on a PDP-12 does not require modification to run on a PDP-8. The FUNCTN argument controls the enabling of all Schmitt triggers, clock-controlled A/D conversions, and clock rate or external input according to the scheme shown in Table 13-3.

Table 13-3
CLOCK Subroutine FUNCTN Arguments

Value of FUNCTN	Effect
0	none, or enable clocked A/D conversion, more than one channel
1	enable Schmitt trigger 1
2	enable Schmitt trigger 2
4	enable Schmitt trigger 3
8	enable clocked A/D conversion, one channel
16	enable the clock to run under external input

Combinations of the conditions in Table 13-3 may be enabled by setting FUNCTN to a value equal to the sum of the values of the desired conditions. For example, to enable all Schmitt triggers, set FUNCTN=7 (the sum of 4, 2, and 1); to enable clocked A/D conversion at an external rate, set FUNCTN=24, etc. If you do not specify a clock condition, the clock is disabled. Every call to CLOCK clears any functions that you may have enabled by previous calls to CLOCK and redefines clock conditions according to the new arguments. If the FUNCTN argument is out of range

R(B) = base rate - maximum number in the set (100000, 10000, 1000, 100) that satisfies the condition. R(B)/R(R)<4096

If you specify an externally driven clock, RATE is interpreted as the number of external ticks between clock interrupts; it must be in the range [1, 4096]. If the argument is outside this range, the interrupt rate will be arbitrary. The RATE argument is actually an overflow count, and the actual rate of the clock can be determined from:

RA = RE/RATE

where RE is the rate of the external input and RA is the actual clock rate. The advantage of an externally driven clock is that it may run at an arbitrarily high rate; however, specifying too high a rate may hang up the FORTRAN system. The calling sequence to define an external clock for the KW12A differs from that of a call for the DW8ES in that the KW12A calling program must enable Schmitt trigger 1. You can obtain optional clock execution on a KW12A external clock when RATE=1. Note that the arguments for a KW12A external clock are sufficient to enable a DK8ES external clock, but not vice versa.

13.1.23 CLOG** (Complex Natural Logarithm Function)

CLOG calculates and returns the natural logarithm of its complex argument, as defined by the relation:

```
LOG(X+iY) = LOG(X**2+Y**2)+i*ATAN(Y/X)
```

13.1.24 CLRPT* (Clear Plot Subroutine)

The calling sequence:

```
CALL CLRPLT (N,BUFFER)
```

clears the current plot, if any, and assigns an N element buffer (designated BUFFER) which will hold 3N/2 points for display. The display is actually created by the PLOT subroutine. The variable BUFFER must be an array with at least N elements.

13.1.25 CMPLX** (Real-to-Complex Conversion Function)

CMPLX accepts two real arguments and returns a complex value with real part equal to the first argument and imaginary part equal to the second argument.

13.1.26 CONJG** (Complex Conjugate Function)

CONJG calculates and returns the complex conjugate of its complex argument. This is accomplished by leaving the real part of the argument unchanged and negating the imaginary part.

13.1.27 COS (Single-Precision Cosine Function)

COS calculates and returns the cosine of a real argument (in radians) by applying the identity:

$$COS(X) = SIN(X+\pi/2)$$

13.1.28 COSD (Single-Precision Cosine in Degrees)

COSD calculates and returns the cosine of a real argument (in degrees). This is accomplished by adding 90 to the argument, converting the result to radians, and extracting the sine.

13.1.29 COSH (Single-Precision Hyperbolic Cosine Function)

COSH calculates and returns the hyperbolic cosine of a real argument according to the relation:

$$\text{If } |x| \leq 88.029$$
$$\text{COSH}(x) = 1/2 \left(\text{EXP}(x) + \frac{1.0}{\text{EXP}(x)}\right)$$

$$\text{If } |x| > 88.028 \text{ and } |x| - \log_e 2 \leq 88.028$$
$$\text{COSH}(x) = \text{EXP}(|x| - \log_e 2)$$

$$\text{If } |x| - \log_e 2 > 88.028$$
$$\text{COSH}(x) = 377737777777_8$$

and an error message is returned.

13.1.30 CPUT (Character Put Subroutine)

The calling sequence:

```
CALL CPUT(STRING,N,CHAR)
```

causes CPUT to insert CHAR as the Nth character in STRING, where STRING is a character string stored in A6 format, and CHAR is a number in the range [0, 63] interpreted as a character. The following program illustrates the use of CGET and CPUT.

```
      DATA STR/'HEY!'/
      WRITE(4,100) STR
100   FORMAT('   HEY! IN ASCII  ',A6)
      WRITE(4,101)
101   FORMAT('   HEY! IN DECIMAL')
      DO 10 I=1,4
      CALL CGET(STR,I,ICHAR)
      WRITE(4,102) ICHAR
 10   CONTINUE
102   FORMAT(I6)
      DO 20 I=1,6
      J=2*I
      CALL CPUT(STR,I,J)
 20   CONTINUE
      WRITE(4,103) STR
103   FORMAT('   NEW STRING   ',A6)
      CALL EXIT
      END

.R F4
*TCHRC/G$
  HEY! IN ASCII  HEY!
  HEY! IN DECIMAL
     8
     5
    25
    33
  NEW STRING   BDFHJL
```

13.1.31 CSIN** (Complex Sine Function)

CSIN calculates and returns the sine of a complex argument according to the relation:

SIN(X+iY) = SIN(X)*COSH(Y)+i*COS(X)*SINH(Y)

13.1.32 CSQRT** (Complex Square Root Function)

CSQRT calculates and returns the square root of a complex argument.

13.1.33 DABS** (Double-Precision Absolute Value Function)

DABS returns the absolute value of its double-precision argument by negating the argument if it is negative, or returning it intact if it is positive.

13.1.34 DATAN** (Double-Precision Arc-Tangent Function)

DATAN calculates and returns the primary arc-tangent of its double-precision argument. The argument is first reduced to the interval [0,] with the identities:

ATAN(-X) = -ATAN(X)

if X>1.0,ATAN(X) = π/2-ATAN(1/X)

if 0.5<X<1.0, ATAN(X) = ATAN(1/2)+ATAN$\left(\frac{2X-1}{X+2}\right)$

The arc-tangent is then calculated as a continued fraction approximation.

13.1.35 DATAN2** (Double-Precision Arc-Tangent of Two Arguments)

DATAN2 accepts two double-precision arguments, one of which is assumed to be an abscissa and the other an ordinate. It calculates the arc-tangent of the quotient of the first argument divided by the second argument. The result is then adjusted, depending upon the quadrant in which a point defined by the arguments falls, in the same manner as for the ATAN2 function.

13.1.36 DATE (OS/8 Date Subroutine)

DATE accepts three integer arguments, accesses the current OS/8 system date, and returns an integer from 1 to 12 corresponding to the current month as the first argument, an integer from 1 to 31 corresponding to the current day as the second argument, and an integer from 1970 to 1977 corresponding to the current year as the third argument.

13.1.37 DBLE** (Single-to-Double Precision Conversion)

DBLE accepts a real argument and returns a double-precision value equal to the argument, filled out with zeros in the low-order three words.

13.1.38 DCOS** (Double-Precision Cosine Function)

DCOS calculates and returns the cosine of a double-precision argument (in radians). This is accomplished by adding PI/2 to the argument and passing this result to the DSIN function.

13.1.39 DEXP** (Double-Precision Exponential Function)

DEXP calculates and returns the exponential function of its double-precision argument by applying the method of Kogbetliantz (IBM Journal of Research and Development, April, 1957, pp 110-115).

13.1.40 DIM (Single-Precision Positive Real Difference)

DIM calculates and returns the positive difference of two real arguments. That is, if the first argument is larger than the second argument, DIM returns the difference between the arguments; if the first argument is less than or equal to the second argument, DIM returns 0.0.

13.1.41 DLOG** (Double-Precision Natural Logarithm)

DLOG calculates and returns the natural (Naperian) logarithm of its double-precision argument. This is accomplished by reducing the range of the argument through application of a method described by Ralston and Wilf in their text, Numerical Methods for Digital Computers, and then performing a Taylor series expansion.

13.1.42 DLOG10** (Double-Precision Common Logarithm)

DLOG10 calculates and returns the common (base 10) logarithm of its double-precision argument by extracting the natural logarithm and executing a change of base.

13.1.43 DMAX1** (Double-Precision Maximum Value)

DMAX1 accepts an arbitrary number of double-precision arguments and returns the largest of the arguments.

13.1.44 DMIN1** (Double-Precision Minimum Value)

DMIN1 accepts an arbitrary number of double-precision arguments and returns the smallest of the arguments.

13.1.45 DMOD** (Double-Precision A Modulo B Function)

DMOD accepts two double-precision arguments and returns a double-precision value equal to the remainder when the first argument is divided by the second argument. If the second argument is not sufficiently large to prevent overflow, an error message and a value of 0.0 are returned.

13.1.46 DSIGN** (Double-Precision Transfer-of-Sign)

DSIGN accepts two double-precision arguments, calculates the absolute value of the first argument, and returns this value if the second argument is positive (or zero), or the negative of this value if the second argument is negative.

13.1.47 DSIN** (Double-Precision Sine Function)

DSIN calculates and returns the sine of a double-precision argument (in radians). The argument is first reduced to the range [0, PI/2], and the sine is then calculated from a Taylor series approximation.

13.1.48 DSQRT** (Double-Precision Square Root)

DSQRT calculates and returns the (positive) square root of a positive double-precision argument. Any negative argument results in an error message.

13.1.49 EXP (Single-Precision Exponential Function)

EXP calculates and returns the exponential function of a real argument. The algorithm uses a numerical method after Kogbetliantz (IBM Journal of Research and Development, April, 1957, pp 110-115).

13.1.50 EXTLVL* (Read PDP-12 External Level)

EXTLVL accepts two integer, real, or logical arguments. The first argument is assumed to be a PDP-12 external-level number in the range [0, 12]. If the referenced external level is at +3 volts (floating), the second argument is set equal to 0. If the referenced external level is at 0 volts (ground), the second argument is set equal to 1. If the first argument is outside the range [0, 12], the value returned in the second argument is unpredictable. If EXTLVL is called on a PDP-8, the second argument will always be set to zero.

13.1.51 FLOAT (Integer-to-Floating Point Conversion)

FLOAT accepts an integer argument and returns a real variable equal to the argument.

13.1.52 IABS (Integer Absolute Value Function)

IABS calculates and returns the absolute value of an integer variable by leaving the variable unchanged if it is positive (or zero), and negating the variable if it is negative.

13.1.53 IDIM (Integer Positive Difference Function)

IDIM calculates and returns the positive difference of two integer arguments. That is, if the first argument is larger than the second argument, IDIM returns the difference between the arguments; if the first argument is less than or equal to the second argument, IDIM returns a value of 0.

13.1.54 IDINT (Double-Precision Integer Truncation)

IDINT accepts a double-precision argument and returns the largest integer that is less than or equal to the argument.

13.1.55 IFIX (Single-Precision Floating Point-to-Integer)

IFIX is a floating-point truncation function. Given a real argument, it truncates the fractional part of the argument and returns the integral part as an integer. IFIX, AINT, and INT perform the same function.

13.1.56 INT (Single-Precision Floating Point-to-Integer)

INT is a floating-point truncation function that performs the same function as AINT and IFIX.

13.1.57 ISIGN (Integer Transfer of Sign Function)

ISIGN accepts two integer arguments, calculates the absolute value of the first argument, and returns this value if the second argument is positive (or zero), or the negative of this value if the second argument is negative.

13.1.58 LSW* (Read PDP-12 Left Switch Register)

LSW accepts two real, integer or logical arguments. The first argument is assumed to be a PDP-12 left switch register switch number in the range [0, 11]. Upon return, the second argument is set to the logical value of the referenced switch (either 0 or 1). If the first argument is outside the range [0, 11], the result that will be returned in the second argument is unpredictable. If LSW is called on a PDP-8, a value of 0 is always returned.

13.1.59 MAX0 (Single-Precision Maximum Value)

MAX0 accepts an arbitrary number of integer arguments and returns an integer result equal to the largest of the arguments.

13.1.60 MAX1 (Single-Precision Maximum Value)

MAX1 accepts an arbitrary number of real arguments and returns an integer result equal to the largest of the arguments.

13.1.61 MIN0 (Single-Precision Minimum Value Function)

MIN0 accepts an arbitrary number of integer arguments and returns an integer value equal to the smallest of the arguments.

13.1.62 MIN1 (Single-Precision Minimum Value Function)

MIN1 accepts an arbitrary number of real arguments and returns an integer value equal to the smallest of the arguments.

13.1.63 MOD (Integer A Modulo B Function)

MOD accepts two integer arguments and returns an integer value equal to the remainder when the first argument is divided by the second argument. If the second argument is not sufficiently large to prevent overflow, an error message and a value of 0 are returned.

13.1.64 ONQB (Place Task on Background Job Chain)

ONQB is a subroutine that you call from PDP-8 mode RALF code to place a PDP-8 mode task on the list of background tasks. These background tasks are executed in round-robin order whenever the PDP-8 processor has nothing to do (e.g., while waiting for terminal input). If FPP-12 hardware is present, these background subroutines execute in parallel with the execution of the FORTRAN program by the FPP-12. You call ONQB by a sequence such as:

```
          JMS%    XONQB+1
          ADDR    BRJOB

          EXTERN  ONQB
XONQB,    ADDR    ONQB
```

where BRJOB is the address of the background job, a subroutine that must obey all the conventions of ONQI. ONQB resides in field 1 and should only be called from field 1.

13.1.65 ONQI (Place Interrupt Handler on Skip Chain)

ONQI is a subroutine that you call from PDP-8 mode RALF code to put the interrupt handler of a device on the interrupt skip chain. When an interrupt is received by the PDP-8 processor, the processor checks

each device on the skip chain, then the FPP, then the standard FORTRAN peripherals, e.g., the line printer. If a device with a handler on the skip chain causes the interrupt, the PDP-8 processor branches to the handler. You call ONQI by a sequence such as:

```
          JMS%    XONQI+1
          IOT
          ADDR    IHNDLR

XONQI,    ADDR    ONQI
          EXTERN  ONQI
```

where IOT is the actual IOT code for the device skip-on-flag instruction and lHNDLR is the address of the interrupt handler for this device. ONQI always resides in field 1 and must be called by PDP-8 mode RALF code in field 1 only. You enter the interrupt handler with the AC cleared and the data and instruction fields set to 1. It should return with these registers in the same state. You should not call ONQI more than once for any given IOT.

13.1.66 PLOT* (Display Data on PDP-12 or LAB-8/E Scope)

The calling sequence:

```
CALL PLOT (M,X,Y)
```

plots M points -- whose X coordinates are in the array X and whose Y coordinates are in the array Y -- into the plot buffer specified by the CLRPLT routine. A background task plots the contents of all points entered into the plot buffer on the scope whenever the PDP-8 processor would otherwise be idle. When X is 1, X and Y are interpreted as scalars. The scope is scaled with (0,0) in the lower left corner and (1.3,1.0) in the upper right corner. You may alter these values by a call to SCALE.

13.1.67 PLOTR* (Change Scope Buffer Values)

The calling sequence:

```
CALL PLOTR (M,X,Y,I)
```

alters the M entries in the plot buffer beginning at the Ith entry; the new X coordinates are obtained from the array X and the new Y coordinates from the array Y. Calling this subroutine does not alter the number of points displayed by the background display task.

13.1.68 RCLOSE* (Close a PDP-12 Relay)

RCLOSE accepts an integer, real, or logical argument assumed to be a PDP-12 relay number in the range [0, 5] and closes the referenced relay. If the argument falls outside the specified range, the result is unpredictable. RCLOSE has no effect when called on a PDP-8.

13.1.69 REAL** (Complex-to-Real Conversion Function)

REAL accepts a complex argument and returns a real value equal to the real part of the argument.

13.1.70 REALTM* (Buffered/Clocked Sampling)

REALTM performs buffered/clocked sampling on the PDP-12 or LAB-8/E. The calling sequence is:

```
CALL REALTM (BUFFER,LENGTH,CSTART,NCHANL,NPTS)
```

where

BUFFER is an array to be used by REALTM as a ring buffer

LENGTH is the size of BUFFER

CSTART is the first channel to sample at each clock interrupt (0-15)

NCHANL is the number of channels to sample at each time step (If NCHANL = 1, then argument 1 of the call to CLOCK may specify clock-initiated A/D sampling (eight images). If NCHANL>1, then argument 1 of CLOCK CALL should not specify clock-initiated sampling. Fetching of the first sample will be initiated in the clock interrupt routines, or 50-100 s after the clock tick. The other samples are taken as soon as possible, about 100-200 s later for each sample.)

NPTS is the total number of samples to take

Algorithm and Comments

The following program samples 500 points from channel 3 at 10 Hz and plots them on the scope:

```
          DIMENSION PLTBUF(400),DATBUF(50)
1         CALL CLRPLT(400,PLTBUF)
          CALL REALTM (DATBUF,50,3,1,500)
          CALL CLOCK (8,10)
          DO 100 I=1,500
100       CALL PLOT(1,I/384.,ADB(X)/1024.+.5)
C         NOW PAUSE SO THAT POINTS WILL BE DISPLAYED
          READ(1,10)Q
10        FORMAT(I2)
          GO TO 1
          STOP
          END
```

13.1.71 ROPEN* (Open a PDP-12 Relay)

ROPEN accepts one integer, real, or logical argument, assumed to be a PDP-12 relay number in the range [0, 5], and opens the referenced relay. If the argument falls outside the specified range, the result is unpredictable. ROPEN has no effect when called on a PDP-8.

13.1.72 RSW (Read Switch Register)

RSW accepts two real, integer, or logical arguments. The first argument is assumed to be a switch register switch number in the range [0, 11]. The second argument is set to the logical value of the referenced switch (right switch register on the PDP-12). If the first argument falls outside the range [0, 11], the result that will be returned in the second argument is unpredictable.

13.1.73 SCALE* (Define Scale of Scope)

SCALE defines the scope screen scaling for calls to PLOT. The calling sequence is:

```
CALL SCALE (XLO,YLO,XHI,YHI)
```

where

```
XLO  is the value at the left edge of the screen
YLO  is the value at the bottom of the screen
XHI  is the value at the right edge of the screen
YHI  is the value at the top of the screen
```

If you never call SCALE, the assumed values are equivalent to:

```
CALL SCALE (0,0,1.3,1.0)
```

13.1.74 SIGN (Single-Precision Transfer-of-Sign)

SIGN accepts two real arguments, calculates the absolute value of the first argument, and returns this value if the second argument is positive (or zero), or the negative of this value if the second argument is negative.

13.1.75 SIN (Single-Precision Sine Function)

SIN calculates and returns the sine of a real argument (in radians). The argument is reduced to the first quadrant, and the sine is then computed from a Taylor series expansion.

13.1.76 SIND (Single-Precision Sine (Degrees) Function)

SIND calculates and returns the sine of a real argument (in degrees). This is accomplished by converting the argument to radians and passing this value to the SIN function.

13.1.77 SNGL** (Double-to-Single Precision Conversion)

SNGL accepts a double-precision argument, truncates the low-order bits, and returns the resulting real value.

13.1.78 SINH (Single-Precision Hyperbolic Sign)

SINH calculates and returns the hyperbolic sine of a real argument according to the relations:

$$\text{If } 0.10 < |x| < 87.929, \text{SINH}(x) = 1/2\left[\text{EXP}(x) - \frac{1}{\text{EXP}(x)}\right]$$

$$\text{If } |x| \leq 0.10, \text{SINH}(x) = x + x^{3/6} + x^{5/120}$$

$$\text{If } |x| > 88.028, \text{SINH}(x) = [\text{EXP}(|x| - \log_e 2)] \cdot [\text{signum}(x)]$$

13.1.79 SQRT (Single-Precision Square Root Function)

SQRT calculates and returns the (positive) square root of a positive real argument. Any negative argument results in an error message.

13.1.80 SSW* (Read PDP-12 Sense Switch)

SSW accepts two real, integer, or logical arguments. The first argument is assumed to be a PDP-12 sense switch number in the range [0, 5]. The second argument is set to the logical value of the referenced sense switch. If SSW is called on a PDP-8, a value of zero is always returned. If the first argument falls outside the range [0, 5], the result that will be returned in the second argument is generally unpredictable. The exception is the calling sequence:

```
CALL SSW (14,RUA12)
```

which returns RUA12=0 on a PDP-8 and RUA12=1 on a PDP-12.

13.1.81 SYNC* (Read a Schmitt Trigger)

SYNC determines whether a Schmitt trigger has been fired; you must not call SYNC unless you have called CLOCK at least once. SYNC accepts two real, integer, or logical arguments. The first argument is assumed to be a Schmitt trigger number in the range [1, 3]. The second argument is set to one if the referenced Schmitt trigger has fired since the last time it was read, or to zero otherwise. The referenced Schmitt trigger is also reset to the not-fired, or zero, state. A call to CLOCK sets all triggers to the zero state, and any trigger that was not enabled by a call to CLOCK is always in the zero state. If the first argument falls outside the range [1, 3], an unpredictable result (either zero or one) is generally returned. If the first argument is zero, however, a value of zero is always returned.

13.1.82 TAN (Single-Precision Tangent Function)

TAN calculates and returns the tangent of a real argument (in radians). This is accomplished by computing the quotient of the sine of the argument divided by the cosine of the argument; thus, if the cosine of the argument is zero, an error message is returned.

13.1.83 TAND (Single-Precision Tangent, Degrees)

TAND calculates and returns the tangent of a real argument (in degrees). This is accomplished by converting the argument to radians and passing the resulting value to the TAN routine.

13.1.84 TANH (Single-Precision Hyperbolic Tangent)

TANH calculates and returns the hyperbolic tangent of a real argument by computing the quotient of the hyperbolic sine of the argument divided by the hyperbolic cosine of the argument.

13.1.85 **TIME* (Read Time of Day)**

You may call TIME as a subroutine with one real or integer argument, or as a function with a dummy argument. It returns the elapsed time since the clock was started. This result will be in seconds unless the clock is running under external input, in which case it will be in external ticks, with the interval between ticks specified by the clock rate (see CLOCK).

CHAPTER 14

PAPER TAPE LOADING INSTRUCTIONS

You may load the FORTRAN IV system from paper tape using OS/8 EPIC. Of the nine files that make up the system, the following eight are on separate paper tapes.

F4.SV	RALF.SV
PASS2.SV	LOAD.SV
PASS20.SV	FRTS.SV
PASS3.SV	LIBRA.SV

These files may be read in any order. After these tapes have been read, the six tapes that comprise the library (FORLIB.RL) must be read in ascending numerical order. A typical procedure might be:

.R EPIC	Load OS/8 EPIC.
*SYS:<	Designate the device on which the new FORTRAN IV system will be built and mount the F4.SV tape in the reader.
*/Y	Mount the PASS2.SV tape in the reader.
*/Y	Mount the PASS20.SV tape in the reader.
*/Y	Mount the PASS3.SV tape in the reader.
*/Y	Mount the RALF.SV tape in the reader.
*/Y	Mount the LOAD.SV tape in the reader.
*/Y	Mount the FRTS.SV tape in the reader.
*/Y	Mount the LIBRA.SV tape in the reader.
*/Y	Mount the first FORLIB.RL tape in the reader.
END OF TAPE ENTER NEXT END OF TAPE ENTER NEXT END OF TAPE ENTER NEXT END OF TAPE ENTER NEXT END OF TAPE ENTER NEXT *^C	Continue to read the six FORLIB.RL paper tapes in increasing numerical order.

PAPER TAPE LOADING INSTRUCTIONS

If you use a PDP-12 and create OS/8 FORTRAN IV systems from paper tape that require the real-time capabilities of this system, you must assemble the RALF modules containing REALTM, ADB, ADC, PLOT, CLRPLT, and SCALE and then add these modules to the system library. The routines you assemble and insert are contained on three paper tapes. A typical procedure might be as follows.

```
.ASSIGN SYS DEV
.R PIP
*DEV:FILE1.RA<PTR:
^
*DEV:FILE2.RA<PTR:
^
*DEV:FILE3.RA<PTR:
^
*^C

.R RALF
*DEV:FILE3.RL<DEV:FILE3.R

.R RALF
*DEV:FILE2.RL<DEV:FILE2.RA

.R RALF
*DEV:FILE1.RL<DEV:FILE1.RA
```

Use OS/8 PIP to read the RALF modules, in ascending numerical order, onto temporary files.

Assemble the temporary files under RALF.

CHAPTER 15

FORTRAN IV PLOTTER ROUTINES

The X,Y plotter routines control an incremental plotter (Calcomp 563, 565, or similar) for use with OS/8 FORTRAN IV. The routines permit the user to generate a wide variety of plotted information, including:

- Labeled axes
- Textual data
- Graphs from data arrays (X and Y), with optional scaling of either array and centered symbols denoting the location of a data point
- Variables from the FORTRAN IV program plotted in F format
- Individual point and vector plotting

You also control:

- Pen position (up or down)
- Origin of plotted information
- Scaling of any plot
- Rotation of text and axes

Table 15-1 summarizes plotter routines and their functions.

Table 15-1
FORTRAN IV Plotter Routines

Name	Function
PLOTS	Initializes all other plotter routines to your hardware configuration.
XYPLOT	Moves pen to specified X,Y location with pen in up or down position; permits origin control.
FACTOR	Scales size of subsequent plotting data.
WHERE	Passes current position and factor to your program.

(continued on next page)

Table 15-1 (Cont.)
FORTRAN IV Plotter Routines

Name	Function
SYMBOL	Prints textual information (such as titles) at any angle and special symbols to indicate a data point.
NUMBER	Prints each digit in a variable, including optional decimal point and truncation.
PSCALE	Defines parameters for axis annotation and size of final plot for data array.
AXIS	Plots an axis, at any angle, including segment markings and title.
LINE	Generates the graph of data in two arrays (X and Y).
PLEXIT	Terminates all plotting.

The system must support any OS/8 FORTRAN IV configuration plus: XY/8e interface for PDP-8/E; or XY interface for PDP-12, 8, or 8/I; and an incremental plotter suitable for one of these interfaces.

The system must have OS/8 (QFS8-A) and OS/8 FORTRAN IV (QF008-AB).

15.1 PLOTTER OPERATION

The plotter permits six basic functions: drum down (+X movement), drum up (-X movement), pen left (+Y movement), pen right (-Y movement), pen up, and pen down. Diagonal movement is accomplished by a combination of pen and drum motion. The plotting increment is a function of the plotter itself, generally .005 or .01 inches. Each line plotted is in this incremental unit. Hence upon very close examination vectors plotted at angles other than multiples of 45 degrees may appear sightly nonlinear. This effect is unnoticeable at normal viewing distances from the plotter where all vectors appear smooth. If you request a vector that exceeds the physical width of the plotter, the pen will move to the physical limit and plot the remaining section at the margin. This may distort subsequent plotting, depending on your sequence of commands. Therefore, be sure the pen is either physically located in a useful position at the start of the plot, or use the plotting commands to monitor its position and prevent such problems.

15.2 PLOTTER COMMANDS

15.2.1 PLOTS

You must call the routine PLOTS once at the start of each plotting program to initialize internal parameters to the current configuration. The call is:

```
CALL PLOTS(X,Y)
```

where

X is the increment size of the plotter in inches; generally .01 or .005 inches

Y is 0 if running on a PDP-8/E, 1 if running on a PDP-8/I, PDP-8, or PDP-12

PLOTS initializes the factor (for overall plot size) to 1 and clears old pen location and origin status. Note that although the plotter may actually move in inches, the code can cause it to behave as if it were millimeters (or any other unit) by including the proper conversion in the FORTRAN code.

15.2.2 XYPLOT

XYPLOT is the routine that actually causes pen and drum movements on the plotter. Routines such as NUMBER and AXIS eventually use XYPLOT. This routine is useful when a plot is to be generated one vector at a time by the user program (rather than saving an array, for example). It also controls the origin, defined as the logical point (0,0) for future plotting.

The call is of the form:

```
CALL XYPLOT (X,Y,I)
```

where

X,Y is the X,Y coordinate in inches to which the pen is to move relative to the most recently established origin point

I is an integer of the set (-3,-2,2,3) that controls pen position and establishes the origin point, as follows:

If I=2, the pen is down during the move.
If I=3, the pen is up during the move.
If I is negative, the pen moves to point X,Y and this point is then established as the current origin point (0,0). (If a value outside this set is called, the pen defaults to down.)

For example:

```
CALL XYPLOT(4,-2,-2)
```

moves from the current position to 4,-2 with the pen down and establishes this location as the origin point (0,0).

```
CALL XYPLOT(-7,3,3)
```

moves the pen in the up position to -7,3. If these two commands are sequential, then this move would be -7 inches of X and +3 of Y, from 0,0 to -7,3.

No single vector can be plotted longer than 4095 plotting increments, or approximately 40.9 inches for a .01 increment plotter or 20.4 for a .005 increment plotter.

15.2.3 FACTOR

You can increase or reduce overall plot size by using the FACTOR routine. The call is:

```
CALL FACTOR(Z)
```

where

Z is the ratio of the desired plot size to the current size. This value is initialized by PLOTS to 1. Calling FACTOR with Z=1 resets the plot to its initial size. Use the absolute value of Z. For example, to double the size of the plot, use CALL FACTOR (2); to halve it, use CALL FACTOR (.5).

15.2.4 WHERE

The WHERE routine passes three values to the user program: current X position, current Y position, and current factor. This routine is most commonly used to determine the current location of the pen in a long plotting sequence, or to calculate a delta X or Y value for the next step in a graph.

The call is:

```
CALL WHERE (X,Y,Z)
```

where

X is set to the current X position

Y is set to the current Y position

Z is set to the current factor

Consider the following example:

```
      CALL PLOTS(.01,1)
      CALL XYPLOT(0,0,-3)
      CALL XYPLOT(-5,3,2)
      CALL WHERE(A,B,C)
      WRITE(4,10)A,B
10    FORMAT(1X,'XVAL=',I3,'YVAL=',I3)
      CALL PLEXIT
      END
```

When this program is run, the statement XVAL=-5YVAL=3 will be printed on device 4.

15.2.5 SYMBOL

The SYMBOL routine has two forms:

- Print any number of letters and symbols
- Print a single character

The available character set for both forms is found in Tables 15-2 and 15-3.

Table 15-2
Special Symbols

SYMBOL	CODE	SYMBOL	CODE	SYMBOL	CODE
	100		106		112
	101		107		113
	102		108		114
	103		109		115
	104		110		116
	105		111		117

Table 15-3
Regular Characters

Symbol	Code	Symbol	Code	Symbol	Code
A	1	V	22	+	43
B	2	W	23	,	44
C	3	X	24	-	45
D	4	Y	25	.	46
E	5	Z	26	/	47
F	6	[	27	0	48
G	7	\	28	1	49
H	8	]	29	2	50
I	9	^	30	3	51
J	10	<-	31	4	52
K	11		32	5	53
L	12	!	33	6	54
M	13	"	34	7	55
N	14	#	35	8	56
O	15	$	36	9	57
P	16	%	37	:	58
Q	17	π	38	;	59
R	18	'	39	<	60
S	19	(	40	=	61
T	20	)	41	>	62
U	21	*	42	?	63

15.2.5.1 **Multiple Characters** - You may combine any of the characters in Table 15-3, except pi, in any order to print titles, legends, labels or the like, using a multiple character call:

```
CALL SYMBOL (X,Y,H,T,A,N)
```

where

X,Y is the coordinate in inches of the lower left corner of the first character to be printed

H is the height in inches of each character (Because characters are considered to be on a 7x7 grid, a multiple of 7 times the increment size is recommended (i.e., a minimum of .07 for .01 increment plotters and .035 for .005 increment plotters). The actual plotting grid occupied by any character is 6x4; the remaining 1x3 is used for spacing between characters.)

T is the text in A or Hollerith format

A is the angle at which the text is to be printed and is specified in degrees from the X axis

N is the number (positive integer) of characters to be plotted and must be greater than 0 and equal to or less than the number of characters in T

For example:

```
DIMENSION TEXT(2)
DATA TEXT/'TEXT EXAMPLE'/
CALL PLOTS(.01,1)
CALL XYPLOT(0,0,-3)
CALL SYMBOL(1,1,.21,TEXT,0,12)
CALL PLEXIT
END
```

will, on a non-PDP8/e machine with a .01 increment plotter, initialize the origin point at the current pen location, move from there to 1,1, and print the 12 characters in TEXT, namely TEXT EXAMPLE, in letters .21 inches high at 0 degrees from the X axis, i.e., parallel to the side of the plotter.

The program above is equivalent to:

```
CALL PLOTS(.01,1)
CALL XYPLOT(0,0,-3)
CALL SYMBOL(1,1,.21,12HTEXT EXAMPLE,0,12)
CALL PLEXIT
END
```

Note that the character pi can only be plotted by a single character command because it has no Hollerith representation.

15.2.5.2 **Single Characters** - You can plot two types of single characters:

- Characters from the available character set listed in Table 15-3
- Special symbols used to denote a data point (listed in Table 15-2)

  The use of special symbols differs from that of other characters in that their starting and terminating point is the center of the character, not the lower left corner. These symbols occupy a 4x4 grid.

The call is:

```
CALL SYMBOL (X,Y,H,I,A,N)
```

where

X,Y is the X,Y coordinate of the lower left corner of a regular character, including pi, or the center for a special symbol

H is the height in inches of the symbol and should be 7 times the increment size for a regular character and 4 times the increment size for a special symbol (i.e., .02 or .04 minimum depending on the plotter)

I is in the range 1-63 for regular characters (Table 15-3) and 100-117 for special symbols (Table 15-2) (If a nonacceptable value is used, SYMBOL prints a space in its place.)

A is the angle in degrees from the X axis at which the character is printed

N is -1 if the pen is to be up during the move to X,Y or -2 if the pen is to be down during the move to X,Y

For example:

```
CALL PLOTS(.01,1)
CALL XYPLOT(0,0,-3)
CALL SYMBOL(-6,2,.35,1,180,-1)
CALL PLEXIT
END
```

This will plot the letter A .35" tall at 180 degrees to the X axis on a PDP-8/E. The pen will be up during the move from 0,0 to -6,2, the lower left corner of the A.

```
CALL PLOTS(.01,1)
CALL SYMBOL(1,4,.20,100,270,-2)
CALL PLEXIT
END
```

This will plot the first special character .2 inches tall centered at point 1,4 at an angle of 270 degrees to the X axis on a non-PDP-8/E. You will be able to see the pen move from its current location to the start of the character (1,4).

15.2.6 **NUMBER**

The NUMBER routine facilitates handling internal format data (floating point). It plots floating-point numbers in a format similar to FORTRAN IV F format. One number at a time is plotted using the call:

```
CALL NUMBER (X,Y,H,Z,A,N)
```

where

X,Y is the coordinate of the lower left corner of the first character of the number

H is the height of each character, preferably 7 times increment size (each number is considered to occupy a 7x7 grid)

Z is the number to be plotted (It may be a real or integer number.)

A is the angle to the X axis at which to plot the number

N is an integer that controls the format of the number Z as follows:

Value of N	Result
0	Z is truncated and plotted as an integer followed by a decimal point
-1	Z is truncated and plotted as an integer
=>1	N digits to the right of the decimal point are plotted. The number is rounded based on the value of the (N+1)th digit.
<-1	N-1 digits are truncated from the integer portion of the number.

Note that the accuracy of the number printed cannot exceed 6 digits. However you may plot up to 19 digits with an expected loss of accuracy. If a bad digit is found in Z, that digit defaults to 0. For Z less than one a leading zero is included. For example:

```
CALL PLOTS(.005,1)
C=0
A=198,678
CALL XYPLOT(0,0,-3)
CALL NUMBER(1,1,.07,A,C,0)
CALL NUMBER(1,2,.07,A,C,-1)
CALL NUMBER(1,3,.14,A,C,-2)
CALL NUMBER(1,4,.14,A,C,2)
CALL PLEXIT
END
```

Statistically the above program will be plotted as follows:

Starting Location	Height (Inches)	Number Plotted	Angle
1,1	.07*6/7	198.	0
1,2	.07*6/7	198	0
1,3	.14*6/7	19	0
1,4	.14*6/7	198.68	0

If the number (Z) is out of range of the acceptable number of characters, including minus sign and decimal point, the message:

```
NUMBER OF DIGITS NOT 1-19
```

is printed on the console device (unit 0).

15.2.7 PSCALE

For many applications, the data to be plotted is scattered irregularly across the total range and in a manner not neatly related to unit (inch) increments. To permit plotting data in a finite (user-specified) length graph with labeled axis, invoke the PSCALE routine to establish two critical plotting parameters -- starting value and scaling increment.

The starting value can be positive or negative and a maximum or minimum. It is the value printed at the starting axis annotation. The scaling increment is the delta value between succeeding axis annotations and is the number of data units per inch of plot, adjusted to 1,2,4,5 or 8 * 10^N.

The starting value and the scaling increment are used by the AXIS and LINE routines to produce a properly annotated axis and a graph whose data includes all points in a user-specified length. PSCALE does no plotting; its use occurs in conjunction with AXIS and/or LINE. It is generally called twice -- once for X (abscissa) values and once for Y (ordinate) values.

The call is:

```
CALL PSCALE(A,L,N,I)
```

where

A is the array containing the data to be plotted (This array must have extra locations at the end in which PSCALE can store the starting value and scaling increment, as explained below.)

L is the length (integer) of the axis that the data is to cover (L must be greater than or equal to 1.)

N is the number of data values in A to be considered (N must be greater than or equal to 1.)

I is the increment between data values to be considered

The first value examined when considering I is always A(1), the next is A(1+I). If I is positive, the calculated starting value will be a minimum value. If I is negative, the calculated starting value will be a maximum, and the scaling increment will be negative.

The calculated starting value is stored at A(N*J+1); the scaling increment is stored at A(N*J+J+1) where J is the absolute value of I. Be sure to dimension A to a length sufficient to include these locations. Consider the data array ARRAY:

Element	Contents
1	.5
2	1
3	.9
4	.9
5	3.4
6	3.2
7	3.9
8	4.5
9	5.2
10	5.9

In the statement:

```
CALL PSCALE (ARRAY,5,5,2)
```

PSCALE will use ARRAY(1), ARRAY(3), ARRAY(5), ARRAY(7), and ARRAY(9) in determining the starting value and scaling increment. For the example above, the scaling increment is 2.0 and the starting value is 0.

If an axis length of less than 1 is supplied, the message:

```
AXIS LENGTH <1
```

is printed on device 0. If all elements of the data array are the same, the message:

```
MAX PT = MIN PT
```

is printed.

15.2.8 **AXIS**

For most graphs, the presence of labeled axes adds significantly to interpreting the data. The AXIS routine draws an axis with labeled tic marks at one-inch intervals and a title or other annotation parallel with and centered to the axis. AXIS must be called separately for an X and a Y axis.

You should determine the starting value and scaling increment discussed in PSCALE before calling AXIS.

The call is:

```
CALL AXIS (X,Y,T,N,L,A,F,D)
```

where

X,Y are the coordinates of the start of the axis, in inches, relative to the current origin. Often when two axes are required, X and Y are 0 for both calls. It is suggested that the physical origin of the axis be at least 1/2" in from any edge of the plotter, as annotation will require that space. This position becomes the new origin for subsequent plotting.

T is the title in Hollerith format. It is printed .14 inches high (dependent on existing user-specified plotting factors) and centered along the axis. If the scaling increment is greater than 99 or less than .01, the notation *10 is added at the end of the title.

N is the number of characters in the title (T) to be printed. Use the sign to specify on which side of the axis the tic marks and their labels are to be: positive means the positive (counterclockwise) side of the axis, negative is the negative or clockwise side. Positive labeling is generally used for Y axes and negative for X axes.

L is the length of the axis in inches. Note that you should not allow this value to exceed the width of the plotter for an axis in that direction. The absolute value of L is used.

A is the angle in degrees at which the axis is to be drawn. X axes are generally at 0 and Y axes at 90.

F is the starting value. You use it as the annotation for the first tic mark. The annotations include two significant places after the decimal point. This value may be determined by PSCALE or supplied by the user. If calculated by PSCALE, F must be the appropriate array element. If you choose to calculate your own starting value and scaling increment, be aware that a tiny F and large D or large F and tiny D do not produce a meaningful graph.

D is the scaling increment between tic mark annotations. It may be determined by PSCALE or by the user. If calculated by PSCALE, D must be the appropriate array element.

For best results axes should be drawn at multiples of 45 degrees (including 0). AXIS uses the routine NUMBER.

15.2.9 **LINE**

You can combine pairs of data points in two arrays by LINE and plot them according to user-specified parameters. You can indicate points to be plotted by a special symbol, connecting them by a continuous line. LINE requires a starting value and scaling increment for each array such as those produced by PSCALE.

The call to LINE is:

```
CALL LINE (A,B,N,I,L,J)
```

where

A is the name of the array whose values are to be the abscissa values

B is the name of the array whose values are to be the ordinate values

(For A and B, the (N*I+1)th element must contain its starting value and the (N*I+I+1)th element must contain its scaling increment, as supplied by the user or PSCALE.)

N is the number of points in each array to be plotted (The same number of points is taken from each array.)

I is the increment at which the data in A and B is collected, i.e., every Ith point is plotted (I must be greater than 0.)

L determines the manner in which the line is plotted, as follows:

If L is positive, each point is connected by a line and a special symbol is plotted at each point.

If L is 0, each point is connected by a line, and no symbols are drawn.

If L is negative, no connecting lines are plotted; each point is indicated by a special symbol.

J is a value between 100 and 117 (see Table 15-2) indicating the special symbol to be used in the plot

The pen should be located at the logical 0,0 position of the graph when a call to LINE is issued. If the preceding plot operation was drawing an axis in the usual manner, the pen should be properly positioned. If I or N is less than or equal to 0, the LINE routine returns without plotting.

15.2.10 PLEXIT

In order to permit the plotting routines to finish completely, call the routine PLEXIT once when all plotting commands have been issued. PLEXIT does a final pen up operation.

15.3 IMPLEMENTING THE PLOTTER ROUTINES

15.3.1 Getting Started

In order for the plotter to interface properly to OS/8 FORTRAN IV, you must make the following patch to the file FRTS.SV. It adds a clear plotter flag IOT to the run-time device initialization chain. The sequence is:

```
.GET SYS:FRTS.SV

.ODT

4020/7000 6502      /User types 4020 / Response
                    /at terminal is 7000.
                    /User types 6502

^C                  Type CTRL/C to exit ODT

.SA SYS:FRTS.SV     Assumes FRTS.SV on SYS Device
```

15.3.2 Adding the Plotting Routines

The FORTRAN plotting routines are supplied as relocatable RALF (.RL) modules that you can either add to the FORTRAN library (FORLIB.RL) or specify explicitly to the loader. To add the files to FORLIB.RL, the procedure is:

```
.R LIBRA
*PLOTLB[3]<FORLIB.RL/Z=40
*PLOTLB<XYPLOT.RL,AXIS.RL,PSCALE.RL,LINE.RL,NUMBER.RL
*^C
```

You may then use PLOTLB by specifying it as a library to the loader or copy it using PIP so that no additional loader specifications are required. If you choose not to add the plotting modules to the library and prefer to specify them to the loader, it is suggested that only the modules required by the FORTRAN program be specified so as not to waste space. In general, if you are employing elaborate overlay schemes, you will not want plotting modules in your library, while if you have shorter programs, you will.

The core requirements to the nearest hundredth location of the files are:

XYPLOT	1000 locations in field 1 and 700 elsewhere (includes FACTOR,PLOTS, WHERE, and PLEXIT)
SYMBOL	500
symbol table	700 (regular and special characters)
NUMBER	1300
PSCALE	1000
AXIS	1500 (requires NUMBER)
LINE	600

Note that the routines PLOTS, XYPLOT, FACTOR, WHERE, PLEXIT, SYMBOL, and the symbol table, including the code in field one, are all loaded if any one of those routines is called.

15.3.2.1 **Loading the Plotter Routines from Paper Tape** - If the relocatable plotter routines are supplied on paper tape, you must load them into mass storage using the program EPIC. Place each tape in the reader before typing the response to the asterisk. The sequence is:

```
.R EPIC
*/O$            /Mount XYPLOT.RL
*/Y             /Mount NUMBER.RL
*/Y             /Mount AXIS.RL
*/Y             /Mount PSCALE.RL
*/Y             /Mount LINE.RL
*^C
```

After you have entered this sequence, the files are on device SYS.

An example combining several of the commands is shown below. This program requests user input of text and then plots it as a spiral.

```
      DIMENSION NAME(30)
      ITTY=4
      WRITE(ITTY,100)
100   FORMAT(1X,'TYPE IN TEXT(30 CHARACTER MAX)')
      READ(ITTY,200)NAME
200   FORMAT(30A1)
      WRITE(ITTY,150)
150   FORMAT(1X,'HOW MANY CHARACTERS DID YOU TYPE IN?')
      READ (ITTY,250)NN
250   FORMAT(I2)
      CALL PLOTS(.01,1)
      CALL XYPLOT(0,-30,-3)
      CALL XYPLOT(10,10,-3)
      RAD=3
      SIZE=.1225*RAD
      SPIR=.995
      CONV=180./3.1415
      ANG=0
      BANG=1.5707
      DO 300 J=1,NN
300   CALL SYMBOL((J-1-NN)*SIZE,RAD,SIZE,NAME(J),ANG,1)
380   DO 400 J=1,NN
      T=2*ATAN(SIZE/(2.*RAD))
      ANG=ANG-T*CONV
      X=RAD*COS(BANG)
      Y=RAD*SIN(BANG)
      BANG=BANG-T
      RAD=RAD*SPIR
      SIZE=.1225*RAD
400   CALL SYMBOL(X,Y,SIZE,NAME(J),ANG,1)
      IF(SIZE-.07)500,500,380
500   CALL PLEXIT
      END
```

The plotter output is shown in Figure 15-1.

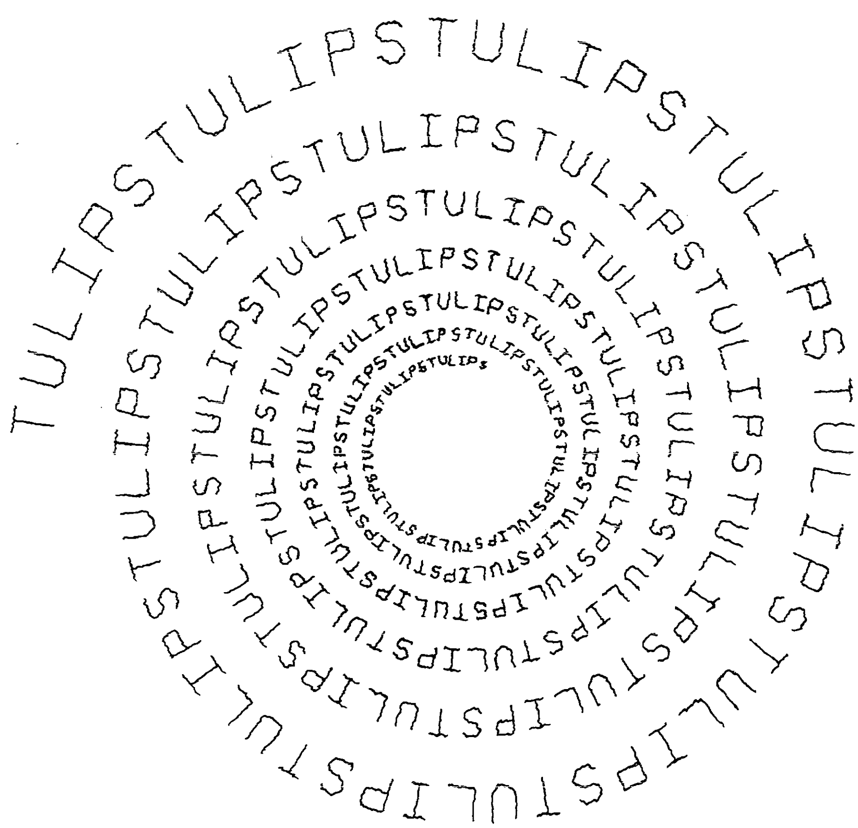


Figure 15-1 Spiral Plotter Example

The next example plots a histogram.

```
      DIMENSION X(50),Y(50),SALES(12),TXT(7)
      DATA SALES/20,30,32,40,50,90,95,90,40,30,20,15/
      DATA TXT/'AIR CONDITIONER SALES EACH MONTH OF 1972 '/
      CALL PLOTS(.01,0)
      CALL XYPLOT(0,-30,-3)
      CALL XYPLOT(1,1,-3)
      DO 5 J=1,12
      I=(J-1)*4+1
      Y(I)=0
      Y(I+1)=SALES(J)
      Y(I+2)=SALES(J)
5     Y(I+3)=0.
      Y(49)=0.
      Y(59)=10.
      DO 10 I=1,48,4
      X(I)=I/4+.75
      X(I+1)=X(I)
      X(I+2)=I/4+.25
10    X(I+3)=X(I+2)
      X(49)=0.
      X(50)=1.
      CALL AXIS(0,0,'MONTH',-5,13,0,0,1)
      CALL AXIS(0,0,'AIR CONDITIONERS (X100)',23,10,90,0,10)
      CALL LINE(X,Y,48,1,0,0)
      CALL SYMBOL(1,10,5,.25,TXT,0,40)
      CALL XYPLOT(12,0,3)
      CALL PLEXIT
      END
```

The plotter output is shown in Figure 15-2.

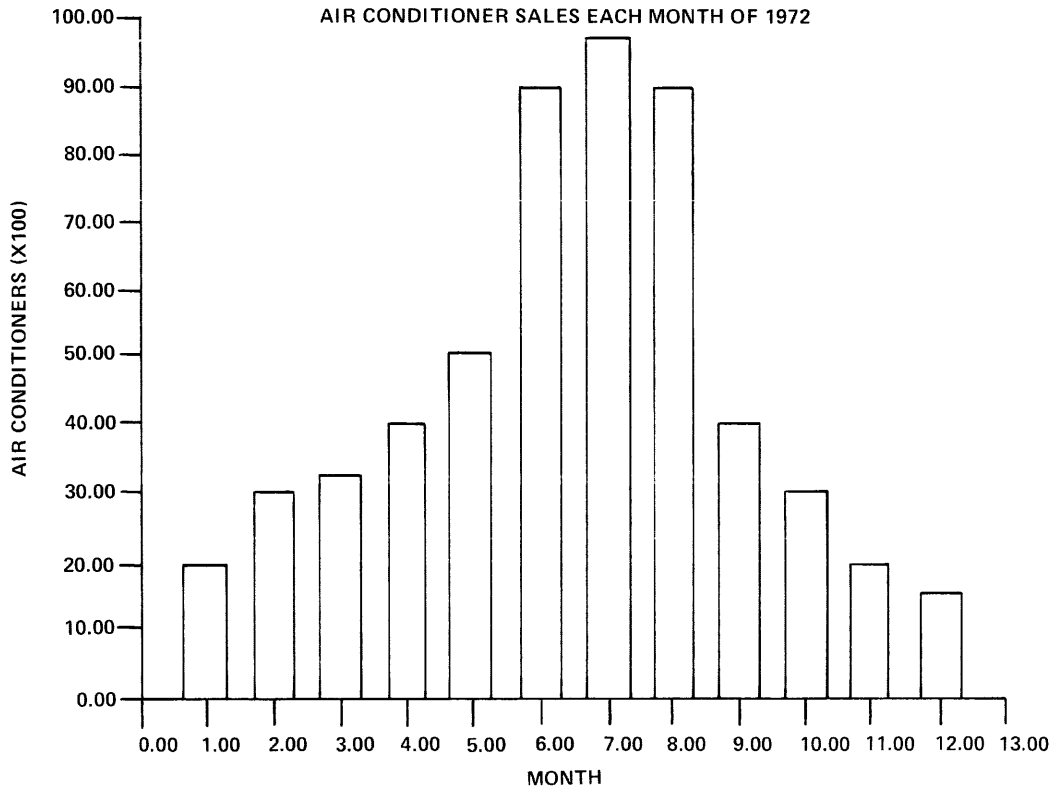


Figure 15-2 Histogram Plotter Example

APPENDIX A

ASCII CHARACTER SET

Decimal Value	ASCII Character	Usage	Decimal Value	ASCII Character	Usage	Decimal Value	ASCII Character	Usage
0	NUL	FILL character	43	+		86	V	
1	SOH		44	,	Comma	87	W	
2	STX		45	-		88	X	
3	ETX	CTRL/C	46	.		89	Y	
4	EOT		47	/		90	Z	
5	ENQ		48	0		91	[	
6	ACK		49	1		92	\	Backslash
7	BEL	BELL	50	2		93	]	
8	BS		51	3		94	^	or ^
9	HT	Horizontal Tab	52	4		95	_	or <
10	LF	Line Feed	53	5		96	`	Grave accent
11	VT	Vertical Tab	54	6		97	a	
12	FF	Form Feed	55	7		98	b	
13	CR	Carriage Return	56	8		99	c	
14	SO		57	9		100	d	
15	SI	CTRL/O	58	:		101	e	
16	DLE		59	;		102	f	
17	DC1		60	<		103	g	
18	DC2		61	=		104	h	
19	DC3		62	>		105	i	
20	DC4		63	?		106	j	
21	NAK	CTRL/U	64	@		107	k	
22	SYN		65	A		108	l	
23	ETB		66	B		109	m	
24	CAN		67	C		110	n	
25	Em		68	D		111	o	
26	SUB	CTRL/Z	69	E		112	p	
27	ESC	Escape*	70	F		113	q	
28	FS		71	G		114	r	
29	GS		72	H		115	s	
30	RS		73	I		116	t	
31	US		74	J		117	u	
32	SP	Space	75	K		118	v	
33	!		76	L		119	w	
34	"		77	M		120	x	
35	#		78	N		121	y	
36	$		79	O		122	z	
37	%		80	P		123		
38	&		81	Q		124	|	Vertical Line
39	'	Apostrophe	82	R		125		
40	(		83	S		126	~	Tilde
41	)		84	T		127	DEL	Rubout
42	*		85	U				

* ALTMODE (ASCII 125) or PREFIX (ASCII 126) keys which appear on some terminals are translated internally into ESCAPE.

APPENDIX B

FORTRAN LANGUAGE SUMMARY

Statement	Form	Effect
Arithmetic	a=b	The value of expression b is assigned to the variable a.
Arithmetic Statement Function Definition	t nam(al...)=x	The value of expression x is assigned to f(al...) after parameter substitution.
ASSIGN	ASSIGN n TO v	Statement number n is assigned as the value of integer variable v for use in an assigned GOTO statement.
BACKSPACE	BACKSPACE u	Peripheral device u is backspaced one record.
BLOCK DATA	BLOCK DATA	Identifies a block data subprogram.
CALL	CALL prog CALL prog(al...)	Invokes subroutine named prog, supply arguments when required.
COMMON	COMMON/block1/a,b../..	Variables (a,b,...) are assigned to a common block.
CONTINUE	CONTINUE	No processing, target for transfers.
DATA	DATA varlist/var/...	Assigns initial or constant values to variables.
DEFINE FILE	DEFINE FILE a(b,c,U,v)	Describes a mass storage file for direct access I/O.
DIMENSION	DIMENSION array (v1...,v7)	Storage allocated according to dimensions specified for the array.

Statement	Form	Effect
DO	DO st l-el,e2,e3	Statements following the DO up to statement st are iterated for values of integer variable i, starting at i=el, incrementing by e3, and terminating when i>e2.
END	END	Cease program compilation; equivalent to STOP in main program or RETURN in subprogram.
END FILE	END FILE u	Writes end-of-file character in file u.
EQUIVALENCE	EQUIVALENCE (vl,v2,...,)	Identifies same storage location for variables within parentheses.
EXTERNAL	EXTERNAL subprogram	Declares a subprogram for use by other subprograms.
FORMAT	FORMAT (specl,spec2,.../...)	Specifies conversions between internal and external representations of data.
FUNCTION	FUNCTION name(al,...)	Indicates an external function definition.
GO TO	(1) GO TO n (2) GO TO (nl,...nk),e (3) GO TO v GO TO v,(nl,...nk)	Transfers control to: (1) statement n (2) to statement nl if e=1, to statement nk if e=k. (3) transfers control to state-number assigned to v optionally checking that v is assigned one of the labels nl,...nk.
IF	IF(arith expr)nl,n2,n3	Transfers control to nl if expr<0, n2 if = 0, or n3 if > 0.
IF	IF(logical expr)st	Executes statement if expression has a value .TRUE., otherwise executes the next statement.
Logical Assignment	v=e	Value of expression E is assigned to variable V.
PAUSE	PAUSE [num]	Program execution interrupted and number printed, if given.

Statement	Form	Effect
READ	READ(u,f) list READ(u,f) READ(u) list READ(a'r) list	Reads a record from a a peripheral device according to specifications given in the argument of the statement.
RETURN	RETURN	Returns control from a subprogram to the calling program.
REWIND	REWIND u	Repositions designated unit to the beginning of the file.
STOP	STOP	Terminate program execution.
SUBROUTINE	SUBROUTINE nam[(al...)]	Declares name to be a subroutine subprogram; al,..., if supplied are dummy arguments.
WRITE	WRITE(u,f) list WRITE(u,f) WRITE(u) list WRITE(a'r) list	Writes a record to a peripheral device according to specifications given in the arguments of the statement.

Operators within each type are shown in order of descending precedence.

Operator		Operand
Arithmetic Type		
**	exponentiation	arithmetic or logical constants, variables, and expressions
*	multiplication	
/	division	
+	addition	
-	subtraction	
Relational Type		
.GT.	greater than	arithmetic or logical constants, variables, and expressions (all relational operators have equal priority)
.GE.	greater than or equal to	
.LT.	less than	
.LE.	less than or equal to	
.EQ.	equal to	
.NE.	not equal to	

Operator		Operand
Logical Type		
.NOT.	.NOT.A is true if and only if A is false	logical constants, variables, and expressions
.AND.	A.AND.B is true if and only if A and B are true	
.OR.	A.OR.B is true if and only if either A or B is true.	
.EQV.	A.EQV.B is true if and only if A and B are both true or A and B are both false.	(equal precedence with .XOR.)
.XOR.	A.XOR.B is true if and only if A is true and B is false or B is true and A is false	(equal precedence with .EQV.)

INDEX

PAL8

CONTENTS

FIGURES

TABLES

1.0 INTRODUCTION

PAL8 is an 8K, two-pass assembler designed to run under the OS/8 Operating System. Pass 1 reads the input file and sets up the symbol table. Pass 2 reads the input file and uses the symbol table created in pass 1 to generate the binary (object) file. The binary file is an absolute binary tape you may load into core with the Absolute Loader or Binary Loader. As an optional third pass, a side-by-side octal and symbolic listing and the symbol table are output. (Using the options available, the three passes may be automatically executed. However, if the source file is to be read from the paper tape reader, you must reload the tape for each pass.) You can use the listing file as an input to the Cross Reference Program (CREF), and you can request the symbol table to be in a form suitable for input to DDT. If you specify a listing file, but not a binary file or /L or /G option, PAL8 does not execute pass 2, but goes directly from pass 1 to pass 3.

PAL8 has pseudo-ops and options not available in the other PDP-8 assemblers and can handle I/O from any OS/8 device that handles ASCII text. It is loaded and saved by way of the OS/8 Monitor and Absolute Loader. It will accept input generated by the Editor and will generate output acceptable to the Absolute Loader and CREF.

2.0 CALLING AND USING PAL8

Call PAL8 from the system device by typing:

```
.R PAL8
```

in response to the Keyboard Monitor dot. The system replies by activating the Command Decoder, which in turn prints an asterisk (*) in the left margin of the teleprinter paper. At this point enter a command string that indicates the binary and listing output devices and file names, the input devices and file names, and any options you select. You may specify 1 to 9 input files. The format of the command string is:

```
*DEV:BINARY,DEV:LISTING,DEV:CREFLS<DEV:INPUT/OPTIONS
```

If you omit the extension to the file name, PAL8 assumes the following extensions:

```
.PA for input file
.BN for binary output file
.LS for listing output file
.TM for intermediate CREF file (if you specified the /C option)
```

A null output file indicates that PAL8 is not to generate an output file of that type. For example, to assemble, load, and run a PAL8 program named PROGRM that is stored on DECtape unit 1, type:

```
.R PAL8
*BIN<DTA1:PROGRM/G
```

After the assembly, PAL8 will load and run the program with the starting address assumed to be location 0200 in field 0, and store the binary on the system device as BIN.BN.

The assembler prints any error messages encountered in the program on the teleprinter. Typing CTRL/O at the keyboard during an assembly will suppress the printing of error messages on the teleprinter; however, the assembler still prints messages in the output file, and they appear immediately before the line that is in error.

3.0 PAL8 OPTIONS

Table 1 lists the options available in PAL8 that can be indicated in the command string typed to the Command Decoder.

When you specify the /L or /G option, you can also include any option to the Absolute Loader in the I/O specification line for PAL8, such as = starting address option. If you do not specify an address, execution begins at 200. If no binary output file is specified with /L or /G a temporary file, PAL8BN.TM, is created and loaded.

Table 1
PAL8 Run-Time Options

Option	Meaning
/B	Make the operator ! a 6-bit left shift instead of an inclusive OR. (A!B equals A^100+B)
/C	Chain to SYS:CREF.SV after assembly. The second output file specified is the output file passed to CREF. The third output file is where PAL8 generates its output. If you give no third output file, SYS:CREFLS.TM is assumed. The /C option supersedes the /G and /L options if specified in the same command string.
/D	Generate a DDT-compatible symbol table (applicable only if a listing file is specified).
/E	Enable error messages if PAL8 generates a link. The LG error message would also be generated.
/F	Disable extra zero fill in TEXT pseudo-op. If the text in the TEXT pseudo-op contains an even number of characters, no word of zeroes will be added to the end.
/G	Call the Absolute Loader, load the binary file, and begin execution at the indicated starting address. If no starting address is indicated, start at 200.
/H	Generate nonpaginated output. Header, page numbers, and page format are suppressed (applicable only if a listing file is specified).
/J	Do not list lines containing code in conditional brackets that is conditionalized out.
/K	Causes systems containing 12K or more of core to use field(s) 2 and up as symbol table storage.
/L	Call the Absolute Loader at the end of the assembly and load the binary file (applicable only if a binary file was specified).
/N	Generate the symbol table, but not the listing (applicable only if a listing file is specified; the /H option is assumed).

(continued on next page)

Table 1 (Cont.)
PAL8 Run-Time Options

Option	Meaning
/O	Disable origining to 200 after pseudo-op. The origin retains the status it had before the FIELD pseudo-op.
/S	Omit the symbol table normally generated with the listing (applicable only if a listing file is specified).
/T	Output a carriage return/line feed in place of the form feed character(s) in the program (applicable only if a listing file is specified).
/W	Do not remember the number of literals that were previously stored on a page after origining off page and then back on again.

3.1 Examples of Specification Strings

Example 1:

```
.R PAL8
*PTP:,LPT:<SOURCE
```

The lines in example 1 command PAL8 to load the assembler from the system device and assemble the program SOURCE.PA (or SOURCE). PAL8 puts the binary output of the assembly onto the paper tape punch and the listing and symbol table onto the line printer.

Example 2:

```
.R PAL8
*,LISTIN<PROG/S
```

The second line of example 2 commands PAL8 to assemble PROG.PA (or PROG), putting the listing only into the file LISTIN.LS on the default device DSK. PAL8 does not generate a binary output nor a symbol table.

Example 3:

```
.R PAL8
*BIN<INPUT.XY/G=600
```

The specification line of example 3 assembles INPUT.XY, putting the binary output into a file named BIN.BN. It then calls the Absolute Loader, which loads the file BIN.BN and starts it at 600. (The equal sign preceding the 600 (=600) is an option to the Absolute Loader specifying the starting address.)

Example 4:

```
.R PAL8
*DTA1:PROG
```

The lines of example 4 will assemble the file PROG from device DTA1 to check for errors, which are listed on the teleprinter. There are no output files.

4.0 RESTARTING AND TERMINATING PAL8

PAL8 may only be restarted if the Command Decoder has not been dismissed. For example:

```
.R PAL8
*^C
.ASSIGN DTA7 DSK
.ST
*
```

If you attempt a restart after you have dismissed the Command Decoder, PAL8 types NO!! and returns control to the Keyboard Monitor. You must call PAL8 for each assembly.

5.0 CHARACTER SET

The following characters are acceptable as input to PAL8:

- Alphabetic characters: A through Z
- Numeric characters: 0 through 9
- Characters described in following sections as special characters and operators
- Characters that are ignored during assembly, such as LINE FEED, FORM FEED, TAB, and RUBOUT

All other characters are illegal (except when used in a comment) and cause PAL8 to print the error message:

IC nnnn

during pass 1; nnnn represents the location where the illegal character occurred. (As assembly proceeds, each instruction is assigned a location determined by the current location counter, described in Section 9.3. When an illegal character or any other error is encountered during assembly, the value of the current location counter is returned in the error message.) Illegal characters do not generally cause assembly to halt. If an illegal character occurs in the middle of a symbol, the symbol is terminated at that point.

6.0 STATEMENTS

PAL8 source programs are usually prepared on the console terminal (using the OS/8 EDITOR or TECO) as a sequence of statements. Each statement is written on a single line and is terminated by typing the RETURN key. There are four types of elements in a PAL8 statement you can identify by the order of their appearance in the statement and by the separating (or delimiting) character that follows or precedes the element. These are:

- label
- instruction
- operand
- /comment

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

6.1 Labels

A label is the symbolic name you create to identify the location of a statement in the program. If you are using a label, it must appear first in a statement. It must begin with an alphabetic character, contain only alphanumeric characters, and be terminated by a comma. There must be no intervening spaces between any of the characters and the comma.

6.2 Instructions

An instruction may be one or more of the mnemonic machine instructions or it may be a pseudo-operation that directs assembly processing (see Section 12.0 for a description of assembly pseudo-ops). Terminate instructions with one or more spaces (or tabs if an operand follows) or with a semicolon, slash, or carriage return.

6.3 Operands

Operands are the octal or symbolic addresses of an assembly language instruction or the argument of a pseudo-operator, and can be any expression. In each case, interpretation of an operand depends upon the instruction or the pseudo-op. Terminate operands with a semicolon, slash, or carriage return.

6.4 Comments

You may add notes or comments to a statement by separating them from the remainder of the line with a slash. Such comments do not affect assembly processing or program execution but are useful in the program listing for later analysis or debugging. The assembler ignores everything from the slash to the next carriage return.

It is possible to have only a carriage return on a line, resulting in a blank line in the final listing. PAL8 gives no error message.

7.0 FORMAT EFFECTORS

The following characters are useful in controlling the format of an assembly listing. They allow you to produce a neat, readable listing by providing a means of spacing through the program.

7.1 Form Feed

The form feed code causes the assembler to output blank lines in order to skip to a new page in the output listing during pass 3. This is useful in creating a page-by-page listing. Generate the form feed by typing a CTRL/L on the console terminal.

7.2 Tabulations

You use tabulations in the body of a source program to separate fields into columns. For example, a line written:

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

is more readable if you insert tabs to form:

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

7.3 Statement Terminators

Use the RETURN key to terminate a statement and cause a carriage return/line feed combination to occur in the listing. You may also use the semicolon (;) as a statement terminator. It is considered identical to a carriage return except that it will not terminate a comment. For example:

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

The entire expression between the slash and the carriage return is considered a comment. Thus in this case the assembler ignores the TAD B. If, for example, you wish to write a sequence of instructions to rotate the contents of the accumulator and link six places to the right, it might look like the following:

```
RTR
RTR
RTR
```

However, you can alternatively place all three instructions on a single line by separating them with the special character semicolon and terminating the entire line with a carriage return. You can then write the above sequence of instructions:

```
RTR;RTR;RTR
```

NOTE

If you desire an OS/8 CREF listing, there are certain restrictions on the use of semicolons. Refer to TBS.

These multistatement lines are particularly useful when setting aside a section of data storage for use during processing. For example, you could reserve a four-word cleared block by specifying either of the following:

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

or

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

You may use either format to input data words (data words may be in the form of numbers, symbols, or expressions, explained in the following sections). Each of the following lines generates one storage word in the object program:

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

8.0 NUMBERS

Any sequence of digits delimited by either a SPACE, TAB, semicolon, or carriage return forms a number. PAL8 initially interprets numbers in octal (base 8). You can change to decimal using a special pseudo-operator (explained in Section 12.0). You use numbers in conjunction with symbols to form expressions.

9.0 SYMBOLS

A symbol is a string of alphanumeric characters beginning with a letter and delimited by a nonalphanumeric character. Although a symbol may be any length, PAL8 recognizes only the first six characters. Since PAL8 ignores additional characters, symbols that are identical in their first six characters are considered identical.

9.1 Permanent Symbols

The assembler contains a table (called its permanent symbol table) that lists the symbols for all PDP-8 pseudo-op codes, memory reference instructions, operate and IOT (input/output transfer) instructions. These instructions are symbols that PAL8 has defined permanently and need no further definition by you; they are summarized in Section 19.0. For example:

```
HLT     This is a symbolic instruction to which the assembler
        has assigned the value 7402 and stored in its permanent
        symbol table.
```

9.2 User-Defined Symbols

All symbols the assembler has not defined (and represented in its permanent symbol table) you must define within the source program.

You may use a symbol as a statement label, in which case PAL8 assigns it a value equal to the current location counter. It is called a symbolic address and you can use it as an operand or as a reference to an instruction. You may not use permanent symbols (instructions, special characters, and pseudo-ops) as symbolic addresses. The following are examples of legal symbolic addresses:

```
ADDR,
TOTAL,
SUM,
A1,
```

The following are illegal symbolic addresses:

```
AD>M,       (contains an illegal character)
7ABC,       (first character must be alphabetic)
LA BEL,     (must not contain imbedded spaces)
D+TAG,      (contains a nonalphanumeric character)
LABEL  ,    (must be terminated by a comma with no intervening
             spaces)
```

9.3 Current Location Counter

As PAL8 processes source statements, it assigns consecutive memory addresses to the instructions and data words of the object program.

The current location counter contains the address where PAL8 will assemble the next word of object code. It is automatically incremented each time PAL8 assigns a memory location. A statement that generates a single object program storage word increments the location counter by one. Another statement might generate six storage words, incrementing the location counter by six.

You set or reset the location counter by typing an asterisk followed by the octal absolute address value where the next program word is to be stored. If you do not set the origin, PAL8 begins assigning addresses at location 200.

```
          *300          /SET CURRENT LOCATION COUNTER TO 300

TAG,      CLA
          JMP A
B,        0
A,        DCA B
          .
          .
```

PAL8 assigns the symbol TAG (in the preceding example) a value of 0300, the symbol B a value of 0302, and the symbol A a value of 0303. If you define a symbol more than once in this manner, the assembler will print the illegal definition diagnostic:

ID address

where address is the value of the location counter at the second occurrence of the symbol definition. PAL8 does not redefine the symbol.

(For an explanation of diagnostic messages refer to Section 18.0 on PAL8 Error Conditions.) For example:

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

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

```
ID COUNTE+0001
```

Since PAL8 uses the first pass to define all symbols, the assembler will print a diagnostic during pass 2 if you make reference to an undefined symbol. For example:

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

This would produce the undefined symbol diagnostic:

```
US A+0003
```

9.4 Symbol Table

Initially, the assembler's symbol table contains the mnemonic op-codes of the machine instructions and the assembler pseudo-op codes; this is its permanent symbol table. As the assembler processes the source program, PAL8 adds to the symbol table user-defined symbols along with their binary values. It lists the symbol table in alphabetical order at the end of pass 3.

During pass 1, if the symbol table is full (in other words, there is no more memory space in which to store symbols and their associated values), PAL8 prints the diagnostic that indicates this condition:

```
SE address
```

and returns control to the OS/8 Monitor. If the system contains more than 8K of memory, you may choose the /K option with the Run command, or you may use more address arithmetic to reduce the number of symbols. It is also possible to segment a program and assemble the segments separately, taking care to generate proper links between the segments (see Section 13.0 for link generation and storage). PAL8's symbol capacity is 992 symbols. The permanent symbol table contains 24 pseudo-operations and 71 symbols, leaving space for 897 possible user-defined symbols. Each additional 4K allows 992 new symbols.

Instructions concerning altering the permanent symbol table appear in Section 12.16 if you wish to add instructions more suitable to your programming needs.

9.5 Direct Assignment Statements

You may insert new symbols with their assigned values directly into the symbol table by using a direct assignment statement in the form:

```
SYMBOL=VALUE
```

VALUE may be a number or an expression. No spaces or tabs may appear between the symbol to the left of the equal sign and the equal sign itself. The following are examples of direct assignment statements:

```
A=6
EXIT=JMP I O
C=A+B
```

You should have already defined all symbols to the right of the equal sign. The symbol to the left of the equal sign is subject to the same restrictions as a symbolic address, and PAL8 stores its associated value in your symbol table. The use of the equal sign does not increment the location counter; it is, rather, an instruction to the assembler.

A direct assignment statement may also equate a new symbol to the value assigned to a previously defined symbol. For example:

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

PAL8 enters the new symbol, GAMMA, into your symbol table with the value 17. You may change the value assigned to a symbol as follows:

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

The second line of code shown changes the value assigned to ALPHA from 5 to 7.

You may use symbols defined by use of the equal sign in any valid expression. For example:

```
A, 0
B, 0        *200
A=100               /DOES NOT UPDATE CLC
B=400               /DOES NOT UPDATE CLC
A+B                 /THE VALUE 500 IS ASSEMBLED AT LOC. 200
TAD                 /THE VALUE 1200 IS ASSEMBLED AT LOC. 201
```

If the symbol to the left of the equal sign is in the permanent symbol table, PAL8 will print the redefinition diagnostic:

```
RD address
```

as a warning, where address is the value of the location counter at the point of redefinition. PAL8 will store the new value in the symbol table; for example:

```
CLA=7600
```

will cause the diagnostic:

```
RD+200
```

Whenever you use CLA after this point, it will have the value 7600.

9.6 Symbolic Instructions

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

```
TAD     (a mnemonic machine instruction)
PAGE    (an assembler pseudo-op)
ZIP     (an instruction defined by the user)
```

9.7 Symbolic Operands

Symbols used as operands normally have a value you have defined. The assembler allows symbolic references to instructions or data defined elsewhere in the program. Operands may be numbers or expressions. For example:

```
TOTAL, TAD ACI + TAG
```

A two's complement add the values of the two symbols ACI and TAG (that you have already defined; see Section 10.1). This value is then used as the address of the operand.

9.8 Internal Symbol Representation for PAL8

Each permanent and user-defined symbol occupies four words in the symbol table storage area. A PDP-8 instruction has an operation code of three bits as well as an indirect bit, a page bit, and seven address bits. The PAL8 assembler distinguishes between pseudo-ops, memory reference instructions, other permanent symbols, and user-defined symbols in the symbol table.

10.0 EXPRESSIONS

The combination of symbols, numbers, and certain characters called operators, which cause a system to perform specific arithmetic operations, form expressions. Either a comma, carriage return, or semicolon terminates an expression.

10.1 Operators

There are seven characters in PAL8 that act as operators:

+	Two's complement addition
-	Two's complement subtraction
^	Multiplication (unsigned, 12-bit)
%	Division (unsigned, 12-bit)
!	Boolean inclusive OR
&	Boolean AND
Space (or TAB)	Treated as a Boolean inclusive OR except in a memory reference instruction

Two's complement addition and subtraction appear in detail in Chapter 1 of Introduction to Programming. Refer to that handbook if you want more information. PAL8 makes no checks for overflow during assembly, and any overflow bits are lost from the high-order end. For example:

7755+24 will give a result of 1

You may use the operators + and - freely as prefix operators. PAL8 performs multiplication by repeated addition. It makes no checks for sign or overflow. All 12 bits of each factor are considered as magnitude. For example:

3000^2 will give a result of 6000

PAL8 performs division by repeated subtraction. The number of subtractions PAL8 performs is the quotient. The remainder is not saved and no checks are made for sign. Division by 0 will arbitrarily yield a result of 0. For example:

7000%1000 will yield a result of 7

You could write this as:

-1000%1000

In this case you might expect the answer to be -1 (7777); but because all 12 bits are considered as magnitude, the result is still 7.

Use of the multiplication and division operators requires an attention to sign on your part beyond what is required for simple addition and subtraction. Table 2 contains examples of operators.

The ! operator causes PAL8 to perform a Boolean inclusive OR bit by bit between the left-hand term and the right-hand term. (The inclusive OR is explained in Chapter 1 of Introduction to Programming.) There is an option you can give to the assembler to have ! interpreted as a 6-bit left shift of the left term prior to the inclusive OR of the right. According to this interpretation:

```
if A=1 and B=2
then A!B=0102
```

Table 2
Use of Operators

Expression	Optional Form	Result
7777+2	-1+2	+1
7776-3	-2-3	7773 or -5
0^2		0
2^0		0
1000^7		7000 or -1000
0%12		0
12%0		0
7777%1	-1%1	7777 or -1
7000%1000	-1000%1000	7
1%2		0

Under normal conditions A!B would be 0003. The & operator causes PAL8 to perform a Boolean AND bit by bit between the left and right values. The operation is the same as the memory reference instruction AND indicates.

SPACE has special significance depending on the context in which you use it. When the symbol preceding the space is not a memory reference instruction as in the following example:

```
SMA CLA
```

it causes PAL8 to perform an inclusive OR between the terms of the expression. In this case, SMA=7500 and CLA=7600. The expression SMA CLA is assembled as 7700. When you use SPACE following pseudo-operators it merely delimits the symbol. When you use it after memory reference operators it also signals the assembler that a memory reference instruction must be assembled.

User-defined symbols are treated as operate instructions. For example:

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

Possible expressions and their values follow, using the symbols just defined. Notice that the assembler reduces each expression to one four-digit (octal) word:

```
A       0333
B       0222
A+B     0555
A-B     0111
-A      7445
1-B     7557
B-1     0221
A!B     0333    (an inclusive OR is performed)
-71     7707
```

If you are loading the information generated, the current location counter is incremented. For example:

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

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

```
HLT=HLT CLA
```

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

```
        *4721
TEMP,
TEM2,   0
```

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

Since a PDP-8 instruction has an operation code of three bits as well as an indirect bit, a page bit, and seven address bits, the assembler must combine memory reference instructions somewhat

differently from the way in which it combines operate or IOT instructions. The assembler differentiates between the symbols in its permanent symbol table and user-defined symbols. PAL8 uses the following symbols as memory reference instructions:

AND	0000	Logical AND
TAD	1000	Two's complement addition
ISZ	2000	Increment and skip if zero
DCA	3000	Deposit and clear accumulator
JMS	4000	Jump to subroutine
JMP	5000	Jump

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

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

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

```
A       100 001 000 001
JMP     101 000 000 000
```

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

```
            000 001 000 001
```

PAL8 tests the remaining bits of the address to see if they are zeros (page zero reference); if they are not, the current page bit is set:

```
            000 011 000 001
```

PAL8 then ORs the operation code into the JMP expression to form:

```
            101 011 000 001
```

or, more concisely in octal:

```
    5301
```

In addition to performing the above tests, PAL8 compares the page bits of the address field with the page bits of the current location counter. If the page bits of the address field are nonzero and do not equal the page bits of the current location counter, an out-of-page reference is being attempted and the assembler will take action as described in the section on link generation and storage (Section 13.0).

10.2 Special Characters

In addition to the operators described in the previous section, PAL8 recognizes several special characters that serve specific functions in the assembly process. These characters are:

=	equal sign
,	comma
*	asterisk
.	dot
"	double quote

()	parentheses
[]	square brackets
/	slash
;	semicolon
<>	angle brackets
$	dollar sign

The equal sign, comma, asterisk, slash, and semicolon have been previously described. The remainder will be described next.

The special character dot (.) always has a value equal to the value of the current location counter. You may use it as any integer or symbol (except to the left of an equal sign); you must precede it by a space when you use it as an operand. For example:

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

is equivalent to JMP 0202. Also,

```
*300
.+2400
```

will produce in location 0300 the quantity 2700. Consider:

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

The second line (CALL=JMS I.) does not increment the current location counter; therefore, PAL8 places 0027 in location 2200 and places CALL in your symbol table with an associated value of 4600 (the octal equivalent of JMS I).

If you precede a single character by a double quote ("), PAL8 uses the 8-bit value of ASCII code for the character rather than interpreting the character as a symbol (ASCII codes are listed in Appendix A). For example:

```
CLA
TAD    ("A
```

The constant 0301 is placed in the accumulator. The code:

```
".   /DOUBLE QUOTE AND DOT
```

will be assembled as 0256. The character must not be a carriage return or one of the characters that is ignored on input (discussed at the end of this section).

Left and right parentheses () enclose a current page literal (closing member is optional).

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

The left parenthesis is a signal to the assembler that the expression following is to be evaluated and assigned a word in the constants table of the current page. This is the same table in which PAL8 stores the indirect address linkages. In the above example, the quantity 2 is stored in a word in the linkage and literals list beginning at the top of the current memory page. The instruction in which the literal appears is encoded with an address referring to the address of the literal. PAL8 assigns a literal to storage the first time it encounters it; subsequent reference to that literal from the current page is made to the same register. The use of literals frees symbol storage space for variables and makes programs more readable.

If you wish to assign literals to page zero rather than to the current page, use square brackets, [ ], in place of parentheses. This enables you to reference a single literal from any page of memory. For example:

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

The closing member is optional. Literals may take the following forms: constant term, variable term, instruction, expression, or another literal.

NOTE

You can nest literals; for example:

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

You may continue this type of nesting in some cases to as many as six levels, depending on the number of other literals on the page and the complexity of the expressions within the nest. If you reach the limits of the assembler, the error messages BE (too many levels of nesting) or PE (too many literals) will result.

Use angle brackets as conditional delimiters. The code enclosed in the angle brackets is to be assembled or ignored contingent upon the definition of the symbol or value of the expression within the angle brackets. (Use the IFDEF, IFNDEF, IFZERO, and IFNZRO pseudo-operators with angle brackets. These are described in Section 12.0.)

NOTE

If your program has conditionals, avoid using angle brackets. The brackets may be interpreted as beginning or terminating the conditional.

The dollar sign character ($) is optional at the end of a program and is interpreted as an unconditional end-of-pass. It may however occur in a text string, comment or " term, in which case it is interpreted in the same manner as any other character.

The assembler handles the following characters for the pass 3 listing, but they are otherwise ignored:

FORM FEED	Used to skip to a new page
LINE FEED	Used to create a line spacing without causing a carriage return
RUBOUT	Used by the EDITOR to allow corrections in the input file

Nonprinting characters include:

SPACE
TAB
RETURN

11.0 INSTRUCTIONS

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

11.1 Memory Reference Instructions

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

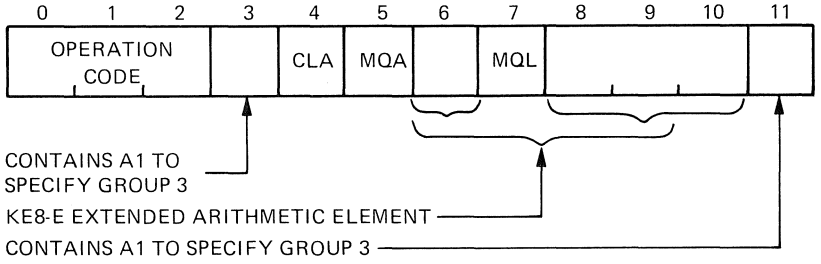


Figure 1 Memory Reference Bit Instructions

Bits 0 through 2 contain the operation code of the instruction PAL8 is to perform. Bit 3 tells the computer if the instruction is indirect. Bit 4 tells the computer if the instruction is referencing the current page or page zero. This leaves bits 5 through 11 (7 bits) to specify an address. These 7 bits can specify 200 octal (128 decimal) locations; the page bit increases accessible locations to 400 octal or 256 decimal. A list of the memory reference instructions and their codes is given in Section 19.0.

In PAL8 a memory reference instruction must be followed by a space(s) or tab(s), an optional I or Z designation, and any valid expression. It may be defined with the FIXMRI instruction (see Section 12.16, Altering the Permanent Symbol Table). You may define permanent symbols using the FIXTAB instruction and use them in address fields as follows:

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

11.2 Indirect Addressing

When the character I appears in a statement between a memory reference instruction and an operand, PAL8 interprets the operand as the address (or location) containing the address of the operand you are using in the current statement. Consider:

```
TAD 40
```

which is a direct address statement, where 40 is the location on page zero containing the quantity to be added to the accumulator. You may make direct reference to locations on the current page and page zero. For compatibility with older paper tape assemblers the symbol Z is also accepted as a way of indicating a page zero reference, as follows:

```
TAD Z 40
```

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

```
TAD I 40
```

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

NOTE

> Because the letter I indicates indirect addressing, do not use it as a variable. Likewise you should never use the letter Z as a variable because it is sometimes used to indicate a page zero reference.

11.3 Microinstructions

Microinstructions are divided into two groups: operate instructions and Input/Output Transfer (IOT) microinstructions. Operate microinstructions are further subdivided into Group 1, Group 2, and Group 3 designations.

NOTE

If you mistakenly specify an illegal combination of microinstructions, the assembler will perform an inclusive OR between them; for example:

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

11.3.1 **Operate Microinstructions** - Within the operate group, there are three groups of microinstructions that you cannot mix. Group 1 microinstructions perform clear, complement, rotate, and increment operations; they are designated by a cleared bit 3 of the machine instruction word.

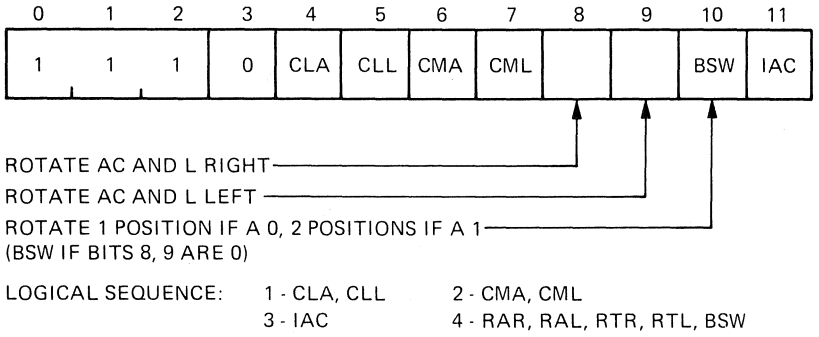


Figure 2 Group 1 Operate Microinstruction Bit Assignments

Group 2 microinstructions first check the contents of the accumulator and link; then, based on the check, they either continue to the next instruction or skip it. You can identify Group 2 microinstructions by a set bit 3 and a cleared bit 11 of the machine instruction word.

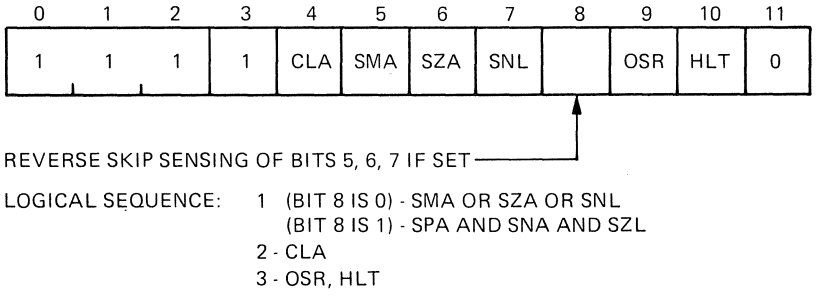


Figure 3 Group 2 Operate Microinstruction Bit Assignments

Group 3 microinstructions reference the MQ register. You can distinguish them from Group 2 instructions because bits 3 and 11 are set. The other bits are part of a hardware arithmetic option.

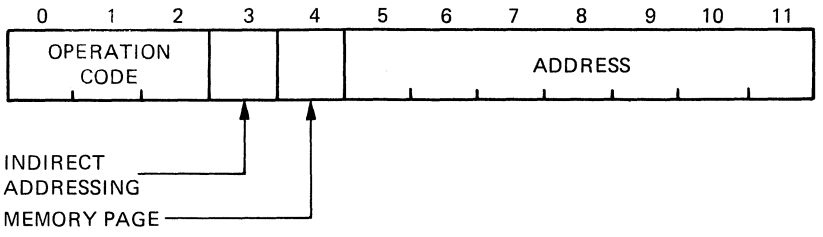


Figure 4 Group 3 Operate Microinstruction Bit Assignments

You cannot combine Group 1 and Group 2 microinstructions since bit 3 determines either one or the other. Within Group 2, there are two groups of skip instructions. You can refer to them as the OR group and the AND group.

OR Group	AND Group
SMA	SPA
SZA	SNA
SNL	SZL

The OR group is designated by a cleared bit 8, and the AND group by a set bit 8. You cannot combine OR and AND group instructions since bit 8 determines either one or the other.

If you do combine skip instructions, it is important to note the conditions under which a skip may occur.

- OR Group--If you have combined these skips in a statement, the inclusive OR of the conditions determines the skip. For example:

  ```
  SZA SNL
  ```

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

- AND Group--If you have combined the skips in a statement, the logical AND of the conditions determines the skip. For example:

  ```
  SNA SZL
  ```

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

11.3.2 **Input/Output Transfer Microinstructions** - If you want to initiate operation of peripheral equipment and effect an information transfer between the central processor and the input/output device(s) -- that is, between the console terminal and the line printer -- use I/O transfer microinstructions.

11.4 Autoindexing

If you are processing large amounts of data, you will find that interpage references are often necessary for obtaining operands. The PDP-8 computers have facilities to make it easy for you to address this data. When one of the absolute locations from 10 to 17 (octal) is indirectly addressed, the contents of the location is incremented before it is used as an address and the incremented number is left in the location. This allows you to address consecutive memory locations using a minimum of statements. You must remember that initially you must set these locations (10 to 17 on page 0) to one less than the first desired address. Because of their characteristics, these locations are called autoindex registers. No incrementation takes place when you address locations 10 to 17 directly. For example, if the instruction to be executed next is in location 300 and the data to be referenced is on the page starting at location 5000, use autoindex register 10 to address the data as follows:

```
0276    1377    TADC4777      /=5000-1
0277    3010    DCA10         /SET UP AUTO INDEX
0300    1410    TADI10        /INCREMENT TO 5000
.       .       .             /BEFORE USE AN AN
.       .       .              ADDRESS
.       .       .
0377    4777    C4777,4777
```

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

12.0 PSEUDO-OPERATORS

You may use pseudo-operators to direct the assembler to perform certain tasks or to interpret subsequent coding in a certain manner. Some pseudo-ops generate storage words in the object program, other pseudo-ops tell the assembler how to proceed with the assembly. Pseudo-ops are maintained in the permanent symbol table. The function of each PAL8 pseudo-op follows.

12.1 Indirect and Page Zero Addressing

Use the pseudo-operators I and Z to specify the type of addressing to be performed. These were discussed in Section 11.2.

12.2 Radix Control

Numbers used in a source program are initially considered to be octal. However, you may change or alternate the radix interpretation by using the pseudo-operators DECIMAL and OCTAL. The DECIMAL pseudo-op interprets all following numbers as decimal until the occurrence of the pseudo-op OCTAL. The OCTAL pseudo-op resets the radix to its original octal base.

12.3 Extended Memory

The pseudo-op FIELD instructs the assembler to output a field setting so that it may recognize more than one memory field. This field setting is output during pass 2 and is recognized by the Absolute Loader, which loads all subsequent information into the field specified by the expression. The form is:

```
FIELDn
```

where n is either an integer, a previously defined symbol, or an expression within the range 0 to 7.

This field setting is output on the binary file during pass 2, followed by an origin setting of 200. As it is executed, the Absolute Loader reads this word and then begins loading information into the new field.

The assembler never remembers the field setting in binary, and no initial field setting is output. However, it appears as the high-order digit of the Location Counter on the listing. A binary file produced without field settings will be loaded into field 0 when using the ABSLDR.

You may use a symbol in one field to reference the same location in any other field. Use of the CDF and CIF instructions determines the field to which it refers. (If you are unfamiliar with the IOT's but wish to use them, refer to the PDP/8E Small Computer Handbook and experiment with several short test programs to learn their effect.) You must use CDF and CIF instructions prior to any instruction referencing a location outside the current field, as shown in the following example:

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

When you use FIELD, the assembler follows the new FIELD setting with an origin at location 200. For this reason, if you want to assemble code at location 400 in field 1 type:

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

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

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

Specifying the /O option to PAL8 inhibits the origin to 200 after a FIELD pseudo-op.

12.4 End-of-File

PAUSE signals the assembler to stop processing the file being read. You should use a PAUSE only at the physical end of a file and with two or more segments of one program. When the assembler reaches a PAUSE statement, it ignores the remainder of the file and continues processing the next input file. In such a case PAUSE must be present or a PH error will occur. The PAUSE pseudo-op is present mainly for compatibility with paper tape assemblers, and its use is optional.

12.5 Resetting the Location Counter

The PAGE n pseudo-op resets the location counter to the first address of page n, where n is either an integer, a previously defined symbol, or a symbolic expression, whose terms have been defined previously and whose value is from 0 to 37 inclusive. If you do not specify n, the location counter is reset to the next logical page of memory. For example:

```
PAGE 2 sets the location counter to 00400
PAGE 6 sets the location counter to 01400
```

If you use the pseudo-op without an argument, the current location counter, if at the first location of a page, will not move. In the following example, the code TAD B is assembled into location 00400:

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

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

12.6 Entering Text Strings

The TEXT pseudo-op allows you to enter a string of text characters as data and store in 6-bit ASCII. To do this, use the pseudo-op TEXT followed by a space or spaces, a delimiting character (which must be a printing character), the string of text, and the same delimiting character. Following the last character, a 6-bit zero is inserted as a stop code. For example:

```
TAG,      TEXT/123*/
```

The string would be stored as:

```
6162
6352
0000
```

The /F option inhibits the generation of the extra 6-bit zero character.

12.7 Suppressing the Listing

Those portions of the source program enclosed by XLIST pseudo-ops will not appear in the listing file; but the code will still be assembled.

You may use two XLIST pseudo-ops to enclose the code to be suppressed. The first XLIST, with no argument, will suppress the listing, and the second XLIST will resume it. XLIST may also be used with an expression as an argument; a listing will be inhibited if the expression is not equal to zero, or allowed if the expression is equal to zero. XLIST pseudo-ops never appear in the assembly listing.

12.8 Reserving Memory

ZBLOCK instructs the assembler to reserve n words of memory containing zeroes, starting at the word indicated by the current location counter. It is of the form:

```
ZBLOCK n
```

For example:

```
ZBLOCK 40
```

causes the assembler to reserve 40 (octal) words. The n may be an expression. If n=0, no locations are reserved.

12.9 Conditional Assembly Pseudo-Operators

The IFDEF pseudo-op takes the form:

```
IFDEF symbol <source code>
```

If you have previously defined the symbol indicated, the code contained in the angle brackets is assembled. If you have not defined the symbol, this code is ignored. You may contain any number of statements or lines of code in the angle brackets. The format of the IFDEF statement requires a single space before and after the symbol.

The IFNDEF pseudo-op is similar in form to IFDEF and is expressed:

```
IFNDEF symbol <source code>
```

If the symbol indicated has not been previously defined, the source code in angle brackets is assembled. If the symbol is defined, the code in the angle brackets is ignored.

The IFZERO pseudo-op is of the form:

```
IFZERO expression <source code>
```

If the evaluated (arithmetic or logical) expresssion is equal to zero, the code within the angle brackets is assembled. If the expression is non-zero, the code is ignored. You may contain any number of statements or lines of code in the angle brackets. The expression may not contain any imbedded spaces and must have a single space preceding and following it. IFNZRO is similar in form to the IFZERO pseudo-op and is expressed:

```
IFNZRO expression <source code>
```

If the evaluated (arithmetic or logical) expression is not equal to zero, the source code within the angle brackets is assembled. If the expression is equal to zero, this code is ignored. You can nest pseudo-ops, for example:

```
IFDEF SYM <IFNZRO X2 <...> >
```

In order to include or delete statements, PAL8 evaluates the outermost pseudo-op first.

```
IFZERO     A<
.
.
(code)
.
.
>...
```

12.10 Use of Conditionals

You can construct useful pseudo-ops such as IFNEG and IFPOS as in the following example:

```
IFNEG expression <statements>     (assemble statements
                                   if expression is
                                   negative)
```

can be written as:

```
IFNZRO expressions &4000 <statements>
```

while its complement

```
IFPOS expression <statements>
```

can be implemented by writing:

```
IFZERO expression &4000 <statements>
```

To prevent PAL8 from printing nonsatisfied condition assembly statements in the listing, use the following solution, employing complementary conditionals:

```
IFNDEF  LTAPE <XLIST>
IFDEF   LTAPE <
      HERE
      GOES
      THE
      CODE>
IFNDEF LTAPE <XLIST>
```

12.11 Controlling Binary Output

NOPUNCH causes the assembler to cease binary output but continue assembling code. It is ignored except during pass 2.

ENPUNCH causes the assembler to resume binary output after NOPUNCH; it is ignored except during pass 2. For example, you might use these two pseudo-ops when several programs share the same data on page zero. When these programs are to be loaded and executed together, only one page zero need be output.

12.12 Controlling Page Format

The EJECT pseudo-op causes the listing to jump to the top of the next page. A page eject is done automatically every 55 lines; EJECT is useful if you want more frequent paging. If you follow this pseudo-op with a string of characters, the first 50 (octal) characters of that string will be used as a new header line.

12.13 Typesetting Pseudo-Operator

Use DTORG in typesetting to output a two-frame DECtape block number (four digits) in the binary tape. The form of this pseudo-op is as follows:

```
DTORG expression
```

The first frame on the binary tape includes channels 7 and 8 punched (in the same manner as a FIELD setting) as a signal to a special typesetting loader that it is to load the following data into DECtape block n. The DTORG setting is added into the checksum, unlike the FIELD setting, which is not included. Do not use DTORG and FIELD in the same program.

12.14 Calling OS/8 User Service Routine

You may use the pseudo-operators DEVICE and FILENAME by issuing calls to the OS/8 User Service Routine, but they have no other meaning to the assembler. The form for these pseudo-ops is:

```
DEVICE name
FILENAME name.extension
```

When you use DEVICE, the name can be from 1 to 4 alphanumeric characters. These are trimmed to 6-bit ASCII and packed into 2 words, filled in with zeroes on the right if necessary. With FILENAME (FILENA is also acceptable) the name (or name.extension) may be from 1 to 6 alphanumeric characters, and the optional extension may be 1 or 2 characters. The characters are trimmed to 6-bit ASCII and packed 2 to a word. Three words are allocated for the filename, filled with

zeroes on the right if fewer than 6 characters are specified, followed by one word for the extension. For example:

```
L,      FILENAME ABC.DA
```

is equivalent to the following coding:

```
L,      0102
        0300
        0000
        0401
```

12.15 Relocation Pseudo-Op

At some point, you may want to assemble code at a given location and then move it to another location for execution. This may result in errors unless the relocated code is assembled in such a way that the assembler assigns to symbols their execution-time addresses rather than their load-time addresses. The RELOC pseudo-op establishes a virtual location counter without altering the actual location counter. The line:

```
RELOC expr
```

sets the virtual location counter to expr. The line:

```
RELOC
```

sets the virtual location counter equal to the actual location counter and terminates the relocation section.

Example:

```
            0400            *400
            2000            RELOC2000
02000*      1377    CODE,   TAD (CODE
02001*      3005            DCA5
02177*      2000            PAGE
            0600            RELOC
```

The location marked CODE is loaded into location 400, but the assembler treats it as if it were loading into location 2000. The asterisks after the location values indicate that the virtual and the actual location counters differ for that line of code. RELOC always causes current page literals to be dumped.

12.16 Altering the Permanent Symbol Table

PAL8 contains a table of symbol definitions for the PDP-8 and OS/8 peripheral devices. These are symbols (such as TAD, DCA, and CLA) that are used in most PDP-8 programs. This table is the permanent symbol table for PAL8.

If you purchase one or more optional devices whose instruction set is not defined among the permanent symbols (for example, EAE or an A/D converter), you should add the necessary symbol definitions to the permanent symbol table in every program you assemble.

Conversely, if you need more space for user-defined symbols, you would probably want to delete all definitions except the ones used in your program. For such purposes, PAL8 has three pseudo-ops you can use to alter the permanent symbol table. The assembler recognizes these pseudo-ops only during pass 1. During either pass 2 or pass 3 they are ignored and have no effect.

EXPUNGE deletes the entire permanent symbol table, except pseudo-ops.

FIXTAB appends all presently defined symbols to the permanent symbol table. All symbols defined before the occurrence of FIXTAB become part of the permanent symbol table for the current assembly.

To append the following instructions to the symbol table, generate an ASCII file called SYM.PA containing:

```
MUY=7405    /MULTIPLY
DVI=7407    /DIVIDE
CLSK=6131   /SKIP ON CLOCK INTERRUPT
FIXTAB      /SO THAT THESE WON'T BE
            /PRINTED IN THE SYMBOL TABLE
```

You then enter the ASCII file in PAL8's input designation. You may also place the definitions at the beginning of the source file. This eliminates the need to load an extra file. Each time you load the assembler, PAL8 restores its permanent symbol table.

The third pseudo-op used to alter the permanent symbol table in PAL8 is FIXMRI. You use FIXMRI to define a memory reference instruction, and it is of the form:

```
FIXMRI name=value
```

Follow the letters FIXMRI with one space, the symbol for the instruction to be defined, an equal sign, and the value of the symbol. The symbol will be defined and stored in the symbol table as a memory reference instruction. You must repeat the pseudo-op for each memory reference instruction you are defining. For example:

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

When the preceding program segment is read into the assembler during pass 1, PAL8 deletes all symbol definitions and adds the three symbols listed to the permanent symbol table. Notice that CLA is not a memory reference instruction. You can perform this process to alter the assembler's symbol table so that it contains only those symbols used at a given installation or by a given program. This may increase the assembler's capacity for user-defined symbols in the program.

13.0 LINK GENERATION AND STORAGE

In addition to handling symbolic addressing on the current page of memory, PAL8 automatically generates links for off-page references. If your program makes reference to an address not on the page where an instruction is located, the assembler sets the indirect bit (bit 3) and an indirect address linkage will be generated on the current memory page. If the off-page reference is already an indirect one, the error diagnostic II (illegal indirect) will be generated. For example:

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

In the example above, the assembler will recognize that the register labeled A is not on the current page (in this case 2600 to 2777) and will generate a link to it as follows:

1. In location 2600 the assembler will place the word 5777, which is equivalent to JMP I 2777.

2. In address 2777 (the last available location on the current page) the assembler will place the word 2117 (the actual address of A).

During pass 3, an apostrophe (') will follow the octal code for the instruction to indicate that a link was generated.

Although the assembler will recognize and generate an indirect address linkage when necessary, you may indicate an explicit indirect address by the pseudo-op I. The assembler cannot generate a link for an instruction that is already specified as being an indirect reference. In this case, the assembler will print the error message II (illegal indirect). For example:

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

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

Literals and links are stored on each page starting at page address 177 (relative) and extending toward page address 0 (relative). Whenever the origin is then set to another page, the literal buffer for the current page is output. This does not affect later execution. Except for page zero, where there is room for 160 (octal) literals and links, each page of memory has room for 100 (octal) literals. Literals and links are stored only as far down as the highest instruction on the page. Further attempts to define literals will result in a PE (page exceeded) or ZE (page zero exceeded) error message.

14.0 CODING PRACTICES

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

- A title comment begins with a slash at the left margin.
- Pseudo-ops may begin at the left margin. However, they are often indented one tab stop to line up with the executable instructions.
- Address labels begin at the left margin. They are separated from succeeding fields by a tab stop.
- Instructions, whether or not they are preceded by a label field, are indented one tab stop.
- A comment is separated from the preceding field by one or two tabs (as required) and a slash; if the comment occupies the whole line it usually begins with a slash at the left margin.

15.0 PROGRAM PREPARATION AND ASSEMBLER OUTPUT

The following program was generated using the OS/8 EDITOR and was assembled with PAL8.

```
/SAMPLE PAL8 PROGRAM
/GETS INPUT FROM KBD,HALTS WHEN "E" IS TYPED
        *200
BEGIN,  KCC
        KSF
        JMP .-1         /WAIT FOR FLAG
        KRB             /READ IN CHARACTER
        TAD (-"E
        SNA CLA         /IS IT E?
        HLT
        JMP BEGIN+1
/END OF EXAMPLE
        $
```

The program consists of statements and pseudo-ops and is terminated by the dollar sign ($). If the program is large, you can segment it by placing it into several files; this often facilitates the editing of the source program since each section will be smaller.

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

The assembler reads the source file for pass 1 and defines all symbols used. During pass 2, the assembler reads the source file and generates the binary code using the symbol table equivalences defined during pass 1. You may load the binary file that is output with the Load command. This binary file consists of an origin setting and data words.

During pass 3, the assembler reads the source file and generates the code from the source statements. The assembly listing is output in ASCII code. It consists of the current location counter, the generated code in octal, and the source statement. Unless you have chosen options to suppress paging or to change the header, the first 50 (octal) characters of the first line of the source program will be used as a heading for each page, followed by the assembler version number, the date and the listing page number. The 5-digit first column is the field number and 4-digit octal address (current location counter); the 4-digit second column is the assembled object code. PAL8 prints the symbol table at the end of the pass. The pass 3 output is:

```
/SAMPLE PAL8 PROGRAM

                /SAMPLE PAL8 PROGRAM
                /GETS INPUT FROM KBD,HALTS WHEN "E" IS TYPED
        0200            *200
00200   6032    BEGIN,  KCC
00201   6031            KSF
00202   5201            JMP .-1             /WAIT FOR FLAG
00203   6036            KRB                 /READ IN CHARACTER
00204   1377            TAD (-"E
00205   7650            SNA CLA             /IS IT E?
00206   7402            HLT
00207   5201            JMP BEGIN+1
              /END OF EXAMPLE
00377   7473
                $

/SAMPLE PAL8 PROGRAM

BEGIN 0200
```

16.0 ABSOLUTE BINARY LOADER

The Absolute Binary Loader is used to load the binary output created by the PAL8 assembler. Input files are loaded according to the options discussed in this section, and a core control block is constructed (see the OS/8 System Reference Manual, Section 3.28, concerning the GET command). The standard input devices are the paper tape reader, DECtape, LINCtape, the default storage device (DSK:), and SYS:, which represents the system device. You may use any other device as an input device if it can contain absolute binary files and if a device handler exists. The terminal (TTY:) should not be used, as the binary code may seem like control characters to the TTY handler.

16.1 Calling and Using ABSLDR

ABSLDR normally accepts absolute binary files (you must load relocatable files with the Linking Loader); however, you can load save (.SV) format files with ABSLDR providing you use the /I option. If you type no extension to the input file name, ABSLDR assumes the .BN extension. Up to nine input files are allowed, but if more than one program is present in a file, only the first program is loaded unless you use the /S option. (This feature allows ABSLDR to ignore any 'noise characters' that might be caused by reading over the end of a paper tape.)

You call the Absolute Binary Loader from the system device by typing:

R ABSLDR

in response to the dot printed by the Keyboard Monitor. The system responds by printing an asterisk at the left margin. The user then types an input line to ABSLDR, indicating input files and any options desired. ABSLDR does not recognize any output files, since the purpose of the loader is to load and optionally start binary output files. The format of the output line is:

*DEV:INPUT.EX/(Options)

When you type the RETURN key at the end of an input specification line, you signal the loader that more input is to be given on the next line. If the ALT MODE key is used as a line terminator, no more input is expected, the Command Decoder is not recalled, and control returns to the Keyboard Monitor. For example:

```
.R ABSLDR
*DTA1:FILE1,FILE2,FILE3,FILE4
*PTR:$
```

The preceding lines cause FILE1, FILE2, FILE3, and FILE4 to be loaded at their absolute locations in core from DECtape 1. A file will then be read from the paper tape reader. The $ character is printed by the ALT MODE key, which indicates a return to the Keyboard Monitor.

NOTE

If the /G option (load and begin execution) is specified, control always passes to the program just loaded, regardless of which line terminator was typed.

When ABSLDR has completed loading and control has returned to the Keyboard Monitor, the program loaded may not be physically in core at that moment. ABSLDR utilizes system scratch blocks to store those locations that would overlay various parts of the Monitor. To examine core locations after using ABSLDR, use ODT (see OS/8 Reference Manual for instructions detailing the use of ODT).

16.1.1 **ABSLDR Options** - The various options accepted by ABSLDR are described in Table 3.

Table 3
ABSLDR Options

Option	Meaning
/8	Used when locations 0-1777 of field 0 are not being used by the program. Eliminates extra DECtape motions to save these locations, hence saves time. See the OS/8 Software Support Manual for details of Job Status Word.
/9	Similar to the /8 option; used when locations 0-1777 of field 1 are not to be saved.

(continued on next page)

Table 3 (Cont.)
ABSLDR Options

Option	Meaning
/I	Treats the input file(s) as a core image file to be overlaid with the input of succeeding lines. (If you do not use this option in the first command line, you can only use it by recalling ABSLDR from the Keyboard Monitor level.) You can use the /I option to make patches to a program you have already saved without reassembling the entire program.
/R	Resets internal core map of ABSLDR to appear as though nothing has been loaded into core.
/S	Loads all binary programs in the specified input file(s) (instead of loading only the first program in each file, which is normally done). The options /S and /I operate on a line-at-a-time basis. Each successive command line must have the option respecified if it is required. For example: *PTR:,,/S *DTA1:A,B,C These command strings instruct ABSLDR to take three files from PTR (loading all binary programs in each file) and three files from DTA1 (loading only the first binary program in each file). The /S option is not implemented on the second line.
/P	Sets bit 3 of the Job Status Word (location 07746) and prevents the Keyboard Monitor from reading a fresh version of the BATCH monitor into core every time the monitor level is reentered from the program level. You can use this option with system programs that never use more than 8K of core (PIP, FORTRAN II, SABR). You should not use the /P option with any program that occupies or modifies core above field 1 (see the BATCH chapter in the OS/8 System Reference Manual for further information).
/G	Starts program execution once the loading procedure is finished. Normally, control returns either to the Monitor or Command Decoder (depending on the terminator key). If /G is specified, control is given to the program just loaded. The starting address is assumed to be 200 unless specified in the input string. Control stays with your program until it is released to the Monitor from within the program. No automatic return to Monitor or the Command Decoder occurs.
/n	Forces loading of all files specified on this input line into field n (where n is an octal integer).
=n	Sets the starting address of the program in core to n, where n is a 5-digit octal integer. ABSLDR inserts a starting address of 0200 in field 0 if you do not indicate any other address. Specifying 0 as a starting address is equivalent to not specifying a starting address, thus ABSLDR would insert a starting address of 0200.

16.1.2 Examples of Input Lines

Example 1:

```
.R ABSLDR
*SYS:PROG.SV/I
*DTA1:PATCH$
.SAVE SYS:PROG
```

The preceding commands load the core image file PROG.SV and then overlay part of that program file with a binary patch from DTA1. Control then returns to Monitor, at which time you save the patched program on the system device.

When you use the /I option, the loader ignores the starting address and Job Status Word of the core image being loaded. You must specify the starting address and contents of the Job Status Word (unless the starting address is 200 in field 0, in which case it need not be specified).

Example 2:

```
.R ABSLDR
*PIP.SV/I
*PTR:=13000(89)$
.SAVE SYS PIP
```

In example 2, you overlay PIP with a binary patch that will not change its starting parameters. You could also accomplish this by using an explicit SAVE:

```
.R ABSLDR
*PIP.SV/I
*PTR:$
.SAVE SYS PIP;13000=6003
```

Example 3:

```
.R ABSLDR
*PTR:(89G)$
```

One binary tape is loaded from the paper tape reader. The program does not use areas 00000-01777 and 10000-11777 of core. The starting address of the program is considered to be 00200; control transfers to your program.

16.2 Notes on Using ABSLDR Correctly

ABSLDR is a complex program that, when used incorrectly, can give unrecoverable errors. Points to remember when using ABSLDR are:

- If you specify an erroneous starting address, control will pass to that address, however random it may be. Thus, specifying a starting address in nonexistent memory, for example, will very likely produce erroneous results and should not be attempted.
- Do not try to load a program into nonexistent memory.

- ABSLDR ignores programs that load into 07600 or 17600. No error is generated, but these locations are never loaded. (It is a good idea not to use 7600 in any field.)
- Do not use old versions of ABSLDR with a new monitor.
- Do not use new versions of ABSLDR with old monitors.

16.3 ABSLDR Error Messages

Table 4 lists the error messages output by ABSLDR. In each case, control returns to the Command Decoder; you may then repeat the entire procedure, resetting the loader (with the /R option) and using different inputs.

Table 4
ABSLDR Error Messages

Message	Meaning
BAD CHECKSUM, FILE # n	File number n of the input file list has a checksum error.
BAD INPUT, FILE # n	Attempt was made to load a nonbinary file as file number n of the input file list, or a non-core image with /I option.
IO ERROR FILE # n	An I/O error has occurred in input file number n.
NO INPUT	No input file was found on the designated device.
NO /I!	Use of /I is prohibited at this point.

17.0 TERMINATING ASSEMBLY

PAL8 will terminate assembly and return to the Monitor under any of the following conditions:

- Normal exit: The end of the source program was reached on pass 2 (or pass 3 if a listing is being generated).
- Fatal error: One of the following error conditions was found and flagged (see Table 5):

  BE DE DF PH SE

- CTRL/C: If you typed it, control returns to the Monitor.

18.0 PAL8 ERROR CONDITIONS

PAL8 will detect and flag error conditions and generate error messages on the console terminal. The format of the error message is:

CODE address

where code is a two-letter code that specifies the type of error, and address is either the absolute octal address where the error occurred or the address of the error relative to the last symbolic label (if there was one) on the current page. For example, because % is an illegal character, the following code:

```
BEG,    TAD LBL
        %TAD LBL
```

would produce the error message:

```
IC BEG+0001
```

The pass 3 listing outputs error messages as two-character messages on the line just prior to the line where the error occurred. The following table lists the PAL8 error codes. Those labeled Fatal Error are followed immediately by an effective CTRL/C.

Table 5
PAL8 Error Codes

Error Code	Meaning
BE	Two PAL8 internal tables have overlapped. You can usually correct this situation by decreasing the level of literal nesting or the number of current page literals used prior to this point on the page. Fatal error: assembly cannot continue.
CF	Chain-to-CREF error. CREF.SV was not found on SYS:.
DE	Device error. An error was detected when trying to read or write a device. Fatal error: assembly cannot continue.
DF	Device full. Fatal error: assembly cannot continue.
IC	Illegal character. The character is ignored and the assembly is continued.
ID	Illegal redefinition of a symbol. An attempt was made to give a previous symbol a new value by means other than the equal sign. The symbol is not redefined.
IE	Illegal equals. An attempt was made to equate a variable to an expression containing an undefined term. The variable remains undefined.
II	Illegal indirect. An off-page reference was made; a link could not be generated because the indirect bit was already set.
IP	Illegal pseudo-op. A pseudo-op was used in the wrong context or with incorrect syntax.
IZ	Illegal page zero reference. The pseudo-op Z was found in an instruction that did not refer to page zero. The Z is ignored.

(continued on next page)

Table 5 (Cont.)
PAL8 Error Codes

Error Code	Meaning
LD	The /L or /G options have been specified and the Absolute Loader is not present on the system.
LG	Link generated. This code is printed only if the /E option was specified to PAL8.
PE	Current non-zero page exceeded. An attempt was made to: 1. Override a literal with an instruction. 2. Override an instruction with a literal. 3. Use more literals than the assembler allows on that page. You can correct a PE situation by decreasing either the number of literals on the page or the number of instructions on the page.
PH	Phase error. A conditional assembly bracket is still in effect at the end of the input stream. This is caused by nonmatching angle bracket (< >) characters in the source file.
RD	Redefinition. A permanent symbol has been defined with =. The new and old definitions do not match. The redefinition is allowed.
SE	Symbol table exceeded. Too many symbols have been defined for the amount of memory available. Fatal error: assembly cannot continue.
UO	Undefined origin. An undefined symbol has occurred in an origin statement.
US	Undefined symbol. A symbol has been processed during pass 2 that was not defined before the end of pass 1.
ZE	Page 0 exceeded. This is the same as PE except with reference to page 0.

19.0 PAL8 PERMANENT SYMBOL TABLE

The following are the most commonly used elements of the PDP-8 instruction set; they are found in the permanent symbol table within the PAL8 Assembler. For additional information on these instructions, and for a description of the symbols used when programming other optional devices, see The Small Computer Handbook, available from the DIGITAL Software Distribution Center. (All times are in microseconds and representative of the PDP-8/E.)

Mnemonic	Code	Operation	Time
Memory Reference Instructions			
AND	0000	Logical AND	2.6
TAD	1000	Two's complement add	2.6
ISZ	2000	Increment and skip if zero	2.6
DCA	3000	Deposit and clear AC	2.6
JMS	4000	Jump to subroutine	2.6
JMP	5000	Jump	1.2
IOT	6000	In/Out transfer	-
OPR	7000	Operate	1.2
Group 1 Operate Microinstructions (1 cycle = 1.2 microseconds)			
NOP	7000	No operation	-
IAC	7001	Increment AC	3
BSW	7002	Byte swap	3
RAL	7004	Rotate AC and link left one	4
RTL	7006	Rotate AC and link left two	4
RAR	7010	Rotate AC and link right one	4
RTR	7012	Rotate AC and link right two	4
CML	7020	Complement the link	2
CMA	7040	Complement the AC	2
CLL	7100	Clear link	1
CLA	7200	Clear AC	1
Group 2 Operate Microinstructions (1 cycle)			
HLT	7402	Halts the computer	3
OSR	7404	Inclusive OR SR with AC	3
SKP	7410	Skip unconditionally	1
SNL	7420	Skip on non-zero link	1
SZL	7430	Skip on zero link	1
SZA	7440	Skip on zero AC	1
SNA	7450	Skip on non-zero AC	1
SMA	7500	Skip on minus AC	1
SPA	7510	Skip on positive AC (zero is positive)	1
Group 3 Operate Microinstructions			
MQA	7501	Multiplier Quotient OR into AC	
MQL	7421	Load Multiplier Quotient	
SWP	7521	Swap AC and Multiplier Quotient	
Combined Operate Microinstructions			
CIA	7041	Complement and increment AC	2.3
STL	7120	Set link to 1	1.2
GLK	7204	Get link (put link in AC, bit 11)	1.4
STA	7240	Set AC to -1	2.0
LAS	7604	Load AC with SR	2.3
Internal IOT Microinstructions			
SKON	6000	Skip with interrupts on and turn them off	
ION	6001	Turn interrupt processor on	1.2
IOF	6002	Turn interrupt processor off	1.2
GTF	6004	Get flags	
RTF	6005	Restore flag, ION	
SGT	6006	Skip if "Greater Than" flag is set	
CAF	6007	Clear all flags	

Mnemonic	Code	Operation	Time
Keyboard/Reader (1 cycle)			
KCF	6030	Clear keyboard flags	
KSF	6031	Skip on keyboard/reader flag	1.2
KCC	6032	Clear keyboard/reader flag and AC; set reader run	1.2
KRS	6034	Read keyboard/reader buffer (static)	1.2
KIE	6035	Set/clear interrupt enable	
KRB	6036	Clear AC, read keyboard buffer (dynamic), clear keyboard flags	1.2
Teleprinter/Punch (1 cycle)			
TFL	6040	Set teleprinter flag	
TSF	6041	Skip on teleprinter/punch flag	1.2
TCF	6042	Clear teleprinter/punch flag	1.2
TPC	6044	Load teleprinter/punch and print	1.2
TSK	6045	Skip on keyboard or teleprinter flag	1.2
TLS	6046	Load teleprinter/punch, print, and clear teleprinter/punch flag	1.2
High-Speed Perforated Tape Reader			
RPE	6010	Set Reader/Punch interrupt enable	1.2
RSF	6011	Skip if reader flag=1	1.2
RRB	6012	Read reader buffer and clear flag	1.2
RFC	6014	Clear flag and buffer and fetch character	1.2
High-Speed Perforated Tape Punch			
PCE	6020	Clear Reader/Punch interrupt enable	1.2
PSF	6021	Skip if punch flag=1	1.2
PCF	6022	Clear flag and buffer	1.2
PPC	6024	Load buffer and punch character	1.2
PLS	6026	Clear flag and buffer, load buffer and punch character	1.2

INDEX

FORTRAN II

CONTENTS

CONTENTS (Cont.)

TABLES

1.0 INTRODUCTION

OS/8 FORTRAN is an improved version of the paper tape 8K FORTRAN. OS/8 FORTRAN contains such added features as Hollerith constants, implied DO loops, chaining, mixing of SABR and FORTRAN statements, and device-independent I/O.

It is assumed that the reader is familiar with the basic concepts of FORTRAN programming. If review is needed, two excellent elementary texts are FORTRAN Programming, by Frederic Stuart, published by John Wiley and Sons, New York, 1969, and A Guide to FORTRAN Programming, by Daniel D. McCracken, published by John Wiley and Sons, New York.

1.1 Calling and Using the OS/8 FORTRAN Compiler

The user calls the FORTRAN compiler by typing:

```
R FORT
```

in reply to the dot generated by the Keyboard Monitor. When the Command Decoder prints an asterisk at the left margin, the user types the appropriate device designations, I/O files, and any of the specification options allowed for 8K FORTRAN. A carriage return is used to terminate a command string and begin compilation.

The line to the Command Decoder consists of 0 to 3 output files, 1 to 9 input files, and any of the available options. The format of the command line is:

```
*BINARY,LISTING,MAP<INPUT/OPTION(S)
```

The first output file holds the binary output in relocatable binary format. If no extension is specified, the extension .RL is assumed. If a binary output file is not indicated in the command line, then no binary output will be generated. (An exception to this occurs when either the /L or /G option is used; this is explained in Table 1.) The second output file contains the listing; if no extension is specified, the extension .LS is assumed. If no listing file is specified, a listing will not be generated. The third output file is the Linking Loader output, and, unless otherwise specified, this file assumes the extension .MP. This output is produced by use of the /M, /U and /P options. (For details on these options and the Linking Loader, see Section 13 in the writeup on SABR in this manual.) 1 to 9 input files are available with OS/8 FORTRAN, although ordinarily only 1 is used. The default extension for input files is .FT.

1.1.1 FORTRAN Options

- Table 1 provides a list of the options which are available under OS/8 FORTRAN. In addition to these, the /N and /S options to the SABR Assembler may be specified to the FORTRAN compiler, and options to the Linking Loader other than /L may also be used.

Table 1
FORTRAN Options

Option	Meaning
/G	Loads and executes the file. The Linking Loader is called, and the binary output file is loaded and executed. (If a binary file is not specified, a temporary file named FORTRL.TM is created and stored on the file device. This file is loaded into core and then deleted from the file device.) If a starting address is not specified (using the options described under the Linking Loader), control is sent to the program entry point MAIN (the FORTRAN compiler gives this name automatically to the main program).
/K	Keeps the file FORTRAN.TM as a permanent file. The FORTRAN compiler produces an output file named FORTRN.TM on the system device. This file, the FORTRAN source program converted into SABR assembly language, serves as input to the 8K SABR assembler, which is automatically called by the compiler. The file FORTRAN.TM is then deleted unless the /K option has been specified. The /K option saves the file as a permanent file, allowing future editing and assembling.
/L	Loads but does not start execution. Calls the Linking Loader at the end of the assembly and loads the specified binary file. (If a binary output file is not specified, then the temporary file FORTRL.TM is loaded into core and deleted from the file device.) When using the /L option, you can terminate the command string by typing either an ALT MODE or a carriage return. If you type ALT MODE, the Loader returns to the Keyboard Monitor with a core image in core; typing the RETURN key instructs the Loader to ask for more input.

1.1.2 **Example Program** - The following example illustrates the ease with which a FORTRAN program can be executed under OS/8. The program TEST has been created with the Symbolic Editor and saved on device SYS:

```
C       FORTRAN DEMO 'TEST'
C       COMPUTE AND PRINT POWERS OF TWO

        DIMENSION A(16)
        WRITE (1,15)
15      FORMAT (/'POWERS OF TWO..EXAMPLE PROGRAM'/)
        DO 20 N=1,16
20      A(N)=2.**N
        WRITE (1,25) (N,A(N),N=1,16)
25      FORMAT ('2** 'I2'='F10.2)
        CALL EXIT
        END
```

The following commands load and execute TEST; execution is automatic with the /G option:

```
.R FORT
*TEST/G

POWERS OF TWO..EXAMPLE PROGRAM

2**   1=       2.00
2**   2=       4.00
2**   3=       8.00
2**   4=      16.00
2**   5=      32.00
2**   6=      64.00
2**   7=     128.00
2**   8=     256.00
2**   9=     512.00
2**  10=    1024.00
2**  11=    2048.00
2**  12=    4096.00
2**  13=    8192.00
2**  14=   16384.00
2**  15=   32768.00
2**  16=   65536.00
```

FORTRAN assembles one main program or subroutine per call. To run a job with multiple subprograms, compile each routine separately and combine them with the Linking Loader.

Typing a CTRL/C (^C) at run time during a non-compute bound job will return control to the Keyboard Monitor. Typing .ST at this point will restart the user's FORTRAN program. If you type ^C when compiling a program, FORTRAN will have to be recalled.

1.1.3 Examples of FORTRAN I/O Specification Commands

Example 1:

```
.R FORT
*DTA1:TEST/G
```

The input file TEST.FT (or TEST) on DTA1 is compiled, the output stored in FORTRN.TM on the system device, and SABR is called. SABR uses FORTRN.TM as input and outputs the assembled file into FORTRL.TM, deleting the old FORTRN.TM. Because the /G option is specified, the Linking Loader then loads FORTRL.TM and the Library Subroutines, deletes FORTRL.TM upon loading, and sends control to the entry point MAIN.

Example 2:

```
.R FORT
*MATRIX<MATRIX.AB/G/U
```

The input file MATRIX.AB on DSK is compiled and the output stored in SYS:FORTRN.TM. SABR is called and assembles SYS:FORTRN.TM, putting the relocatable binary output into DSK:MATRIX.RL and deleting the file FORTRN.MT. Because the /G option is specified, the Linking Loader then loads MATRIX.RL and the Library Subroutines, and then prints on the teleprinter (via /U) a list of undefined external symbols and a count of the unused pages in each memory field.

Example 3:

```
.R FORT
*,,LPT:<INPUT/L/M
*
```

The FORTRAN Compiler compiles and SABR assembles the file DSK:INPUT.FT (or INPUT), outputting the binary file as SYS:FORTRL.TM. The Linking Loader is automatically called (/L) to load SYS:FORTRL.TM into core and delete that file from SYS. The Linking Loader puts a full loading map on the LPT device (/M). The Loader then asks for another command string. If you terminate the line with the ALT MODE key instead of the RETURN key, control is returned to the Keyboard Monitor after loading.

Example 4:

```
.R FORT
*SUB1<SUB1
.R FORT
*SUB2<SUB2
.R FORT
*MAIN/L
*SUB1,SUB2/G
```

The subroutines and the MAIN program are each compiled separately, and the MAIN program is loaded but not executed (as the /L option indicates). The Linking Loader, called at the end of the assembly, waits for more input. The /G option loads the FORTRAN Library Subroutines and initiates execution of the MAIN program.

Example 5:

```
.R FORT
*DTA5:SOURCE/L
```

The file SOURCE on DTA5 is compiled, assembled, and loaded but not executed.

Example 6:

```
.R FORT
*DTA1:PROG,PTP:,PTP:<DTA1:PROG(NMG)
```

If you have a DECtape system, keeping the source program on a non-system DECtape and putting the binary on a non-system DECtape gives the best possible results in terms of minimizing tape motion. The file PROG is loaded and executed. The binary is stored on DTA1 under the name PROG.RL, and the symbol table, the map of the loaded program, and the count of the free pages in each field are punched onto paper tape.

In DECtape systems, you can eliminate excessive DECtape motion by storing LIB8.RL on a non-system tape. Specify to the loader:

```
*DTA2:LIB8.RL/L
```

1.2 Using FORTRAN or SABR with the Interrupt On

SABR code can be run with the interrupt on, provided you have your own interrupt-handling code. That code, which is executed when the interrupt is off, must not call any of the SABR subroutines and must be independent of all SABR or library subroutines and linkage subroutines. With the interrupt on, do not call exit routines or do any generalized (device-independent) I/O, unless those routines are modified to make allowances for interrupts.

1.3 Using PAL8 with SABR or FORTRAN

PAL8 subroutines can be called from a SABR or FORTRAN program. You should build a core image of the running FORTRAN or SABR program and return to the Keyboard Monitor by typing $ (ALT MODE key) on the last Linking Loader Command. Then save the core image. You can use the core image file (.SV) as input to the Absolute Loader (ABSLDR) with the /I option, followed by the binary of the PAL8 routine, for example:

```
.R ABSLDR
*DTA7:CHAIN2.SV/I
*PALSUB.BN/G$
```

The above calls the Absolute Loader, loads the core image CHAIN2.SV and then merges the PALSUB.BN program with it. Execution starts at location 200 and, when completed, the system returns to the Keyboard Monitor for further instructions.

1.4 FORTRAN Data Files

When doing FORTRAN output onto DECtape or disk into a file which is to be read only as a data file by another FORTRAN program, you can save significant time by using the A6 format to output floating-point variables and the A2 format to output integer values. The same format specifications must be used when the data is read. The data file is not an ASCII file and should not be edited with EDIT. The file should only be moved by PIP in image mode (/I option).

Observe the following caution concerning programs which may have been written and compiled with a previous version of OS/8 FORTRAN.

CAUTION

A FORTRAN compiler and its corresponding Library constitute an interlocking set of programs. No user should attempt to compile a program under OS/8 and load it with the paper tape FORTRAN, or vice versa. Similarly, programs developed with the current FORTRAN compiler should not be run under an old FORTRAN system.

2.0 FORTRAN II SOURCE LANGUAGE

2.1 Character Set

The following characters are used in the FORTRAN language.

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

1. The alphabetic characters, A through Z.

2. The numeric characters, 0 through 9.

3. The special characters. (Of these, the characters " ! $ % & # : ? < > ^ [] \ must appear inside FORMAT statement or Hollerith constants.)

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

2.2 FORTRAN Constants

Constants are self-defining numeric values appearing in source statements and are of three types: integer, real, and Hollerith.

2.2.1 **Integer Constants** - An integer (fixed point) constant is 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, for example:

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

2.2.2 **Real Constants** - A real constant is 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 x 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 of + 1.7x10^38.

2.2.3 **Hollerith Constants** - A Hollerith constant is a string of up to 6 characters (including blanks) enclosed in single quotes. A Hollerith constant is treated like a real constant, except that it cannot be used in arithmetic expressions other than for simple equivalence (A=B). Any character except the quote character itself can be used in a Hollerith constant, for example:

```
'MOM'
'A+B=C'
'5 & 10'
```

2.3 FORTRAN Variables

A variable is a named 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. Although any number of characters may be used to make up the variable name, only the first five characters are interpreted as defining the name; the rest are ignored. For example, DELTAX, DELTAY, and DELTA all represent the same variable name.

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.

2.3.1 **Integer Variables** - The name of an integer variable must begin with an I, J, K, L, M, or N. An integer variable undergoes 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 Function Calls) or by an arithmetic statement (see Arithmetic Statements). Integer variables must fall within the range -2047 to +2047.

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 any text on binary arithmetic.

2.3.2 **Real Variables** - A real variable name begins with an alphabetic character other than I, J, K, L, M, or N. Real variables may be converted to integer variables by the function IFIX (see Section 3.7.2.3) or by an arithmetic statement. Real variables undergo no truncation in arithmetic calculations.

2.3.3 **Scalar Variables** - A scalar variable may be either integer or real and represents a single quantity. Examples are as follows:

```
LM
A
G2
TOTAL
J
```

2.3.4 **Array Variables** - An array (subscripted) variable represents a single element of a one- or two-dimensional array of quantities. The array element 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 of one-dimensional arrays are:

```
Y(1)
PORT(K)
```

An example of a two-dimensional array is:

```
A(3*K+2,1)
```

Any array must appear in a DIMENSION statement prior to its 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 (see Storage Allocation). 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)
```

2.3.5 **Subscripting** - Since excessive subscripting tends to use core memory inefficiently, subscripted variables should be used judiciously. For example, the statement:

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

if rewritten as follows would save considerable core memory:

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

2.4 Expressions

An expression is a sequence of constants, variables, and function references separated by arithmetic 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 priority, the calculations are performed from left to right.

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 and are explained later.

Excluding ** (exponentiation), no two arithmetic 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:

1. A real variable may be raised to an integer power:

   ```
   A**2
   ```

2. Mode may be altered by using the functions IFIX and FLOAT (see Function Calls):

   ```
   A*FLOAT(I)
   ```

The I in example 2 above, indicates an integer variable; it is changed to real (in floating point format) by the FLOAT function.

Zero raised to a power of zero yields a result of 1. Zero raised to any other power yields 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 does not cause an error message but uses the absolute value. Thus, the expression (-3.0)**3.0 yields a result of +27.

An arithmetic expression may be enclosed in parentheses and be considered a basic element.

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

An arithmetic expression may consist of a single element (constant, variable, or function call), for example:

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

Compound arithmetic expressions may be formed using arithmetic operators to combine basic elements, for example:

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

Expressions preceded by a + or a - sign are also arithmetic expressions, for example:

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

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

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

This expression is coded as:

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

3.0 FORTRAN STATEMENTS

A FORTRAN source program consists of a series of statements, each of which must start on a separate line. Any FORTRAN statement may appear in the statement field (columns 7 through 72) and may be preceded by a positive number, called a statement number, of from 1 to 4 digits. This statement number serves as an address label and is used when referencing the statement. Statement numbers are coded in columns 1 through 5 of the 72-column line. Statement numbers need not appear in sequential order, but no two statements should have the same number. Statement numbers are limited to a value of 2047 or less.

When you are using the Symbolic Editor to create the source program, typing a CTRL/TAB (generated by holding down the CTRL key and pressing the TAB key) causes a jump over the statement number columns and into the statement field. Except for data within a Hollerith field (see Input/Output Statements, Section 3.4), spaces are ignored by the compiler. You may use spaces freely, however, to make the program listing more readable and to organize data into columns.

3.1 Line Continuation Designator

Statements too long for the statement field of a single terminal line may be continued on the next line. The continued portion must not be given a line number, but must have an alphanumeric character other than 0 in column 6. If you use the Symbolic Editor, you may type a CTRL/TAB followed by a digit from 1 to 9 before continuing the line. The continuation character is not treated as part of the statement.

For example, using spaces, a continued statement would look as follows;

```
      WRITE (3,30)
30    FORMAT (1X,'THE FOLLOWING DATA IS GROUPED INTO THREE
     1 PARTS UNDER THE HEADINGS X, Y, AND Z.')
```

Using tabs, the same statement would be typed:

```
      WRITE (3,30)
30    FORMAT (1X,'THE FOLLOWING DATA IS GROUPED INTO THREE
     1 PARTS UNDER THE HEADINGS X, Y, AND Z.')
```

There is no limit to the number of continuation lines which may appear. However, one restriction is that an implied DO loop must not be broken but must be on one line. For ease in program correction, it is recommended that continuation lines be minimized.

3.2 Comments

The letter C in column 1 of a line designates that line as a comment line. A comment appears in a program listing but has no effect on program compilation. Any number of comment lines may appear in a given program, and comments that are too long for one line may be continued by placing a C in the first column of the next line. A comment line may not appear between another line and its continuation.

FORTRAN statements are of five types:

1. Arithmetic, defining calculations to be performed;
2. Input/Output, directing communication between the program and input/output devices;
3. Control, governing the sequence of execution of statements within a program;
4. Specification, describing the form and content of data within the program;
5. Subprogram, defining the form and occurrence of subprograms and subroutines.

Each of these five types is explained in the following paragraphs.

3.3 Arithmetic Statements

Constants and variables, identified as to type and connected by logical and arithmetic operators, form expressions; one or more expressions form an arithmetic statement. Arithmetic statements are of 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

A=A*B

are quite meaningful and indicate that the value of the variable A is to be changed, for example:

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

The expression value is made to agree in type with the variable before replacement occurs. In the statement:

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

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

3.4 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:

1. Data transmission statements, READ and WRITE, specify transmission of data between computer memory and I/O devices.

2. Nonexecutable FORMAT statements enable conversion between internal data (within core memory) and external data.

3.4.1 **Data Transmission Statements** - The two data transmission statements, READ and WRITE, accomplish input/output transfer of data listed in a FORMAT statement. The two statements are of the form:

```
READ (unit, format) I/O list
WRITE (unit, format) I/O list
```

where unit is a device designation which can be an integer constant or an integer variable, format is a FORMAT statement line number, and the I/O list is a list specifying 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.

An element in an I/O list can take one of the following forms:

1. Arithmetic expression: expressions more complicated than a single variable (which can be subscripted) are meaningless in an input operation.

2. The name of an array (1 or 2 dimensional) : this indicates that every element of the array is to be transmitted. Elements are transmitted in the order in which they are stored in core.

   For example:

```
DIMENSION A(2,2)
READ (1,100) A
```

   reads:

   A(1,1),A(2,1)A(1,2),A(2,2)

3. Implied DO Loops of the form:

   (s(1),s(2),...,s(n),i=m(1),m(2),m(3))

   repeat the list elements (s(n)) with the value of i being equal to m(1) through m(2) having an optional step value of m(3). The m's are integer constants or variables, i is an integer variable, and s(1)-s(n) are the I/O list elements (possibly including an implied DO loop). For example:

```
DIMENSION A(3,6)
WRITE (1,100) I,(A(J,I)J=1,3)
```

   will output the values:

   I,A(1,I),A(2,I),A(3,I)

   When using implied DO loops, remember that the entire implied DO loop must be on the same input line or card. An implied DO loop cannot be continued onto the next line with a continuation character.

If no I/O list is specified for a WRITE statement, then information is read directly from the specified FORMAT statement and written on the device designated.

Data appears on the external device in the form of records. (This should not be confused with the OS/8 record, which is equal to 256(10) words (2 DECtape blocks with the 129th word of each block ignored.)) 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 statement and the I/O list.

Each execution of an I/O 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 below, where 100 is the FORMAT statement referenced:

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

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 statement f.

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

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

3.4.1.1 **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 statement line number, and list is a list of variables whose values are to be input.

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,P
```

3.4.1.2 **WRITE Statement** - The WRITE statement specifies transfer of 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 statement line number, and list is a list of variables to be output.

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

The WRITE statement without a list causes information (generally Hollerith type) to be read directly from the specified format and written on the designated device in ASCII form.

The I/O device designations used in the READ and WRITE statements are described in Table 2.

Table 2
Device Designations

Device Code	Input Designation	Output Designation
1	Teletype keyboard or low-speed reader	Teleprinter
2	High-speed reader	High-speed punch
3	Card reader (CR8/I)	Line printer (LP08)
4	Assignable device*	Assignable device*

*(See Device Independent I/O and Chaining)

If using device code 4, the /I or /O option to the Linking Loader must be given. If the assignable device is a two-page handler, the /H option must be given also.

Device code 3 is assigned to the card reader (for all READ statements), and the line printer (for all WRITE statements). The card reader uses a two-page device handler, which is too large to be used with the device independent I/O feature (device code 4). Therefore, the card reader has its own device code.

The line printer is a separate output device because it can require special formatting, such as inserting a Form Feed to skip to the top of a page. The contents of the first column of any line is a control character. These control characters are never printed. They are as follows:

Character in Column 1	Resulting Spacing
space	single space
0	double space
1	skip to top of next page (Form Feed)
all others	single space

3.4.2 **FORMAT Statement** - The nonexecutable FORMAT statement specifies the form and arrangement of data on the selected external device. FORMAT statements are of the form:

m FORMAT (S(1)S(2),...S(n))

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

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

During transmission of data, the object program scans the designated FORMAT statement; if a specification for a numeric field is present, 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. The FORMAT statement may contain specifications for more items than are indicated by the data transmission statement. The FORMAT statement may also contain specifications for fewer items than are indicated by the data transmission statement, in which case format control reverts to the rightmost left parenthesis in the FORMAT statement. If an input list requires more characters than the input device supplies for a given record, blanks are inserted.

3.4.2.1 **Numeric Fields** - Numeric field specification codes and the corresponding internal and external forms of the numbers are listed in Table 3.

Table 3
Numeric Field Codes

Conversion Code	Internal Form	External Form
E	Binary floating point	Decimal floating point with E exponents: 0.324E+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 code, 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. For example:

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

could be used to output the line

```
   32    -17.60  0.59624575E+03
```

on the output listing.

The field width should always be large enough to include the decimal point, sign, and exponent (plus a leading zero in OS/8 FORTRAN). In all numeric field conversions, if the field width is not large enough to accommodate the converted number, asterisks will be printed; the number is always right-justified in the field.

3.4.2.2 **Numeric Input Conversion** - In general, numeric input conversion is compatible with most other FORTRAN processors. A few exceptions are listed below:

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

   ```
   bbb12
   12bbb
   00012
   ```

   are read as the number 0.12 (where 'b' represents a blank space).

2. A null line delimited by two carriage return/line feed (CR/LF) combinations is treated as a line of blanks, and blanks are appended to the right of a line (if necessary) to fill out a FORMAT statement. Thus:

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

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

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

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

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

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

```
rAw
```

where r is a repetition count, 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 I/O 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. The characters are stored in stripped ASCII format. If not enough data is supplied as input to the variables, the data is padded with blanks on the right, for example:

```
        READ (1,20) M1,M2,M3,M4,M5,M6,M7,M8
20      FORMAT (8A1)
```

If you now type:

123ABC

followed by a carriage return, the variables will have the following values:

Variable	Decimal	Octal	ASCII
M1	-928	6140	1
M2	-864	6240	2
M3	-800	6340	3
M4	96	0140	A
M5	160	0240	B
M6	224	0340	C
M7	-2016	4040	blank
M8	-2016	4040	blank

If the above had been read in the 4A2 format, the values would be as follows:

Variable	Decimal	Octal	ASCII
M1	-910	6162	1 2
M2	-831	6301	3 A
M3	131	0203	B C
M4	-2016	4040	blanks
	.....		
M8	-2016	4040	blanks

Consider a second example:

```
          READ (1,20) ALPHA
20        FORMAT (A6)
```

If you type:

123AB

and a carriage return, the octal value of ALPHA is:

6162 6301 0240

NOTE

The numeric value of alphanumeric characters stored in floating point variables is generally not meaningful.

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

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

nH h(1), h(2),..., h(n)

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

```
FORMAT(17H PROGRAM COMPLETE)
```

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.

An attempt to use H format specifications with a READ statement will cause characters from the format field to be either printed or punched. This feature provides a simple way of identifying data that is to be read from the Teletype keyboard. For example, the following instructions:

```
          READ (1,30)A,B
30        FORMAT (4HA = ,F7.2/4HB = ,F7.2)
```

cause A = and B = to be printed out before the data is read.

By merely enclosing the alphanumeric data in single quotes, you can achieve the same result as in H-conversion; on input, the characters between the single quotes are typed as output 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:

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

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

3.4.2.5 **Blank or Skip Fields** - Blanks can be introduced into an output record or characters skipped on an input record by use of the nX specification. The number n indicates the number of blanks or characters skipped and must be greater than zero. For example:

```
FORMAT(5H STEPI5,10X2HY=F7.3)
```

can be used to output the line:

```
STEP    28           Y=  3.872
```

3.4.2.6 **Mixed Fields** - A Hollerith format field may be placed among other fields of the format. The statement:

```
FORMAT(I5,7H FORCE=F10.5)
```

can be used to output the line:

```
     22 FORCE=  17.68901
```

The separating comma may be omitted after a Hollerith format field, as shown above.

3.4.2.7 **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. The statement:

```
FORMAT(2E12.4,3I5)
```

is equivalent to:

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

3.4.2.8 **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)
```

3.4.2.9 **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 (producing a CR/LF for each record) 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 parenthesis at the end of the format indicate the termination of a record. If the list of an I/O 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(E15.5,315.4),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
```

3.5 Control Statements

The control statements GO TO, IF, DO, PAUSE, STOP, and END alter the sequence of statement execution, temporarily or permanently halt program execution, and stop compilation.

3.5.1 **GO TO Statement** - The GO TO statement has two forms: unconditional and computed.

3.5.1.1 **Unconditional GO TO** - 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.

3.5.1.2 **Computed GO TO** - Computed GO TO statements have the form:

```
GO TO (n(1), n(2),..., n(k)), J
```

where n(1), n(2),..., n(k) are statement numbers and J is a nonsubscripted integer variable. This statement transfers control to the statement numbered n(1), n(2)..., n(k) if J has the value 1, 2,..., k, respectively. The index (J in the above example) of a computed GO TO statement must never be zero or greater than the number of statement numbers in the list (in the example above, not greater than k). 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.

3.5.2 **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 a simple variable or an arithmetic expression.

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

3.5.3 **DO Statement** - The DO statement simplifies the coding of iterative procedures. DO statements are of the form:

```
DO ni = m(1), m(2), m(3)
```

where n is a statement number, i is a scalar 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, terminal, and increment values of the index, for example:

```
DO 10 J=1,N
DO 20 I=J,K,5
DO 30 L=I,J,K
```

The index is incremented and tested before the range of the DO is executed. After the last execution of the range, control passes to the statement immediately following the terminal statement in what is called a normal exit. An exit may also occur by a transfer out of the range taking place before the loop has been executed the total number of times specified in the DO statement.

DO loops may be nested, or contained within one another, provided the range of each contained loop is entirely within the range of the containing DO statement. Nested DO loops may contain the same terminal statement, however. A transfer into a DO loop 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.* The values of the initial, terminal, 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.

3.5.4 **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.

---

* After a normal exit from a DO loop, the index of the DO statement has the value of the index for the final time through the loop plus whatever increment was assigned. For example:

```
DO 10 I=1,5
```

after a normal exit the value of the index is 6. However, it is good programming practice to avoid using the index as a variable following a normal exit until the index has been redefined, as according to ANSI FORTRAN Standards the value is undefined.

3.5.5 **PAUSE, STOP, and END Statements** - The PAUSE and STOP statements affect FORTRAN object program operation; the END statement affects assembler operation only.

3.5.5.1 **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
or      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.

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, follow the PAUSE statement with a call to the OPEN subroutine. The subroutine initializes the I/O devices and sets hardware flags that may have been cleared by pressing the tape feed button, for example:

```
PAUSE
CALL OPEN
```

NOTE

> The CALL OPEN statement in OS/8 FORTRAN also resets all I/O on unit 4, the assignable channel. Any further READs or WRITEs on unit 4 without an intervening IOPEN or OOPEN will print an error message and abort.

3.5.5.2 **Stop Statement** - The STOP statement has the form:

```
STOP
```

It terminates program execution. STOP may occur several times within a single program to indicate alternate points at which execution may cease. Program control is either directed to a STOP statement or transferred around it.

3.5.5.3 **End Statement** - The END statement is of the form:

```
END
```

It signals the compiler to terminate compilation. The END statement must be the last statement of every program. (In OS/8 FORTRAN, the END statement generates a STOP statement as well.)

3.6 Specification Statements

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

3.6.1 **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. Varibles in COMMON statements are assigned to locations in ascending order in field 1 beginning at location 200 storage allocation. The COMMON statement has the general form:

```
COMMON v(1), v(2),..., v(n)
```

where v is a variable name.

3.6.2 **DIMENSION Statement** - The DIMENSION statement is used to declare 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),..., s(n)
```

where s is an array specification. For example:

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

Dimension statements are used for the purpose of reserving sufficient storage space for anticipated data; it is the user's responsibility to see that his subscripting does not conflict with the DIMENSION statement declarations. For example:

```
DIMENSION I(10),J(10),K(10)
I(2,4)=2
J(12)=3
```

The above statements would assemble without error; at run time I(8) would be set equal to 2 and K(2) would be set equal to 3.

NOTE

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

3.6.3 **EQUIVALENCE Statement** - The EQUIVALENCE statement causes more than one variable within a given program to share the same storage location. This is useful when the programmer desires to conserve storage space. The form of the statement is:

```
EQUIVALENCE (v(1), v(2)...),...
```

where v represents a variable name. The inclusion of two or more variables within the parenthetical list indicates that these variables are to share the same memory location and thus have the same value, for example:

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

The variables RED and BLUE are now of equal value. The subscripts of array variables must be integer constants, for example:

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

Because of core memory restrictions within the compiler, variables cannot appear in EQUIVALENCE statements more than once. The following statement is valid:

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

The following statement would not compile correctly:

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

Variables may not appear in both EQUIVALENCE and COMMON statements.

3.7 Subprogram Statements

External subprograms, defined separately from the programs that call them, 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 of times they are used. External subprograms are defined by means of the statements FUNCTION and SUBROUTINE. Functions and subroutines must be compiled independently of the main program and then loaded together with the main program by the Linking Loader.

NOTE

> Care should be exercised when naming a subprogram or subroutine. It must not have the same name as any of the FORTRAN library functions or subroutines, or assembler mnemonics or pseudo-ops, as errors are likely to result. The Library Functions are listed in this chapter, and the symbol table for the SABR Assembler is listed in Appendix C.

Subprogram definition statements may optionally contain dummy arguments representing the arguments of the subprogram. They are used as ordinary identifiers within the subprogram and are replaced by the actual arguments when the subprogram is executed.

3.7.1 **Functions** - A function is called from an arithmetic expression within the main program and returns a single numeric value. A function 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. The FUNCTION statement has the form:

```
FUNCTION identifier (a(1), a(2)..., a(n))
```

where FUNCTION (or FUNC) declares that the program which follows is a function subprogram, and identifier is the name of the function being defined. The identifier must appear as a scalar variable and be assigned a value during execution of the subprogram. This value is the function's value.

Arguments appearing in the list enclosed in parentheses are dummy arguments representing the function arguments. A function must have at least one dummy argument. The arguments must agree in number, order and type with the actual arguments used in the calling program. Function subprograms may be called with expressions and array names as arguments. The corresponding dummy arguments in the FUNCTION statement would then be scalar and array identifiers, respectively. Those representing array names must appear within the subprogram in a DIMENSION statement. Dimensions must be indicated as constants and should be smaller than or equal to the dimensions of the corresponding arrays in the calling program. Dummy arguments to FUNCTION cannot appear in COMMON or EQUIVALENCE statements within the function subprogram.

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 are SUBROUTINE and other FUNCTION statements.

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.

The following short example calculates the gross salary of an individual on the basis of the number of hours he has worked (TIME) and his hourly wage (RATE). The function calculates time and a half for overtime beyond 40 hours. The function name is SUM.

```
      FUNCTION SUM(TIME,RATE)
      IF (TIME-40.) 10,10,20
10    SUM = TIME * RATE
      RETURN
20    SUM = (40.*RATE) + (TIME-40.)*1.5*RATE
      RETURN
      END
```

Depending upon which path the program takes, control will return to the main program at one of the two RETURN statements with the answer. Assume that the main program is set up with a statement to read the employee's weekly record from a list of information prepared on the high-speed reader:

```
READ(2,5) NAME, NUM, NDEP, TIME, RATE
```

This statement reads the person's name, number, department numer, time worked, and hourly wage. The main program then calculates the person's gross pay with a statement such as the following:

```
GROSS = SUM(TIME,RATE)
```

and goes on to calculate withholdings and other payments.

3.7.2 **Subroutines** - A subroutine is called by the main program via a CALL statement. A subroutine may return several or no values. It begins with a SUBROUTINE statement and returns control to the calling program by means of one or more RETURN statements. The SUBROUTINE statement has the form:

```
SUBROUTINE identifier (a(1), a(2)... a(n))
```

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

Subroutines 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.

A subroutine 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 arguments, or it may use arguments to return numbers to the calling program. Subroutines are generally used when the result of a subprogram is not a single value.

Examples of SUBROUTINE statements are as follows:

```
SUBROUTINE FACTO (COEFF,N,ROOTS)
SUBROUTINE RESID (NUM,N,DEN,M,RES)
SUBROUTINE SERIE
```

The only FORTRAN statements not allowed in a subroutine are FUNCTION and other SUBROUTINE statements.

The following short subroutine takes two integer numbers from the main program and exchanges their values. If this exchange of values is to be done at several points in the main program, it is a procedure best performed by a subroutine.

```
SUBROUTINE ICHGE (I,J)
ITEM=I
I=J
J=ITEM
RETURN
END
```

The calling statement for this subroutine might look as follows:

```
CALL ICHGE (M,N)
```

where the values for the variables M and N are to be exchanged.

3.7.2.1 **CALL Statement** - The CALL statement assumes one of two forms:

```
            CALL identifier
     or     CALL identifier (a(1), a(2)..., a(n))
```

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

The arguments (indicated by a(1), through a(n)) 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, for example:

```
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.

3.7.2.2 **RETURN Statement** - The RETURN statement has the form:

```
RETURN
```

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

4.0 FUNCTION LIBRARY

The standard FORTRAN library contains built-in functions, including user-defined functions and subroutines.

Table 4 lists the built-in functions. These are open subroutines: they are incorporated into the compiled program each time the source program names them.

Functions and subroutines are closed routines; their coding appears only once in the compiled program. These routines are entered from various points in a program through jump-type linkages.

Function calls are provided to facilitate the evaluation of functions such as sine, cosine, and square root. A function 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 any arithmetic expression.

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

```
IDENT (ARG,ARG,...,ARG)
```

where IDENT is the identifying function name and ARG is an argument which may be any expression. A function call is evaluated before the expression in which it is contained.

NOTE

A FORTRAN compiler and its corresponding Library constitute an interlocking set of programs. No user should attempt to compile a program under OS/8 and load it with the paper tape FORTRAN, or vice versa. Similarly, programs developed with the current FORTRAN compiler should not be run under an old FORTRAN system.

Table 4
FORTRAN Function Library

Function	Definition	Type of Argument
ABS(x)	the absolute value of x	real
IABS(x)	the absolute value of x	integer
FLOAT(x)	convert x from integer to real format	integer
IFIX(x)	convert x from real to integer format	real
IREM(0)	remainder of last integer divide is returned	integer
IREM(x/y)	remainder of x/y is returned	integer
EXP(x)	exponential of x, e^x	real
ALOG(x)	natural logarithm of x, log(e)^x	real
SIN(x)	sine of x, where x is given in radians	real
COS(x)	cosine of x, where x is given in radians	real
TAN(x)	tangent of x, where x is given in radians	real
ATAN(x)	arc tangent of x, where x is given in radians	real
SQRT(x)	square root of x is returned	real
IRDSW(0)	read the console switch register, returning a decimal equivalent of the octal integer in the switch register. The switch register can be set before executing the FORTRAN program or, using the PAUSE statement, during execution.	integer

5.0 FLOATING POINT ARITHMETIC

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. With the exception of the arctangent function, which is accurate to seven places over the entire range, the results of function operations are accurate to six decimal places.

The floating point arithmetic routines check for both overflow and underflow. Overflow will cause the OVFL error message to be printed, 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.

6.0 DEVICE INDEPENDENT I/O AND CHAINING

OS/8 FORTRAN provides for device-independent, file-oriented, formatted I/O through use of the device number 4 in the READ and WRITE statements and several utility subroutines. These are described below.

6.1 The IOPEN Subroutine

The subroutine IOPEN prepares the system to accept input from a specified device when device code 4 is used in a READ statement. IOPEN takes two arguments which are interpreted as Hollerith strings. After a

```
CALL IOPEN(A,B)
```

any READ statement reading from device 4 will read from the file specified by B (which must have the extension .DA) on the device specified by A. For example, the following statement will prepare for input from the file DTA5:INPUT.DA.

```
CALL IOPEN('DTA5','INPUT')
```

The following statement will prepare for input from the device F1, which, in this case, is a non-file-structured device.

```
CALL IOPEN('F1',0)
```

If the file and device names are input via READ statements which use A format in their FORMAT statements, then A6 format must be used. The sign @, rather than spaces, should be used to fill in empty characters. For example, the following statements are contained in a program:

```
          WRITE (1,20)
20        FORMAT ('ENTER FILE NAME')
          READ (1,22)FNAME
22        FORMAT (A6)
          CALL IOPEN('DSK',FNAME)
            .
            .
            .
```

The Teletype prints:

```
ENTER FILE NAME
```

The user responds:

```
ABC@@@
```

6.2 The OOPEN Subroutine

The subroutine OOPEN prepares the system to send output to a specified device when device code 4 is used in a WRITE statement. The arguments of OOPEN are treated like those of IOPEN. Future WRITE statements using device 4 write on the device and file specified in the call to OOPEN. An error message is printed if the program has previously issued a CALL OOPEN without issuing a subsequent CALL OCLOSE. For example, the following statement prepares device 4 to output on device PTP.

```
CALL OOPEN('PTP',0)
```

The following statement prepares device 4 to output to the file SYS:LADE.DA.

```
CALL OOPEN('SYS','LADE')
```

6.3 The OCLOSE Subroutine

The subroutine OCLOSE is called with no arguments. Its function is to terminate output on the output file opened by OOPEN. If OCLOSE is not called after a file has been written, that output file will never exist on the specified device.

6.4 The CHAIN Subroutine

A call to the subroutine CHAIN terminates execution of the calling program and starts execution of the core image on the system device as specified by the argument to CHAIN. Variables in common storage are not disturbed. For example, the following statement:

```
CALL CHAIN('PROG2')
```

causes the file SYS:PROG2.SV to be loaded and started. Notice that PROG2 must be compiled and stored on the system device as a core image (.SV) file in order to be successfully accessed.

6.5 The EXIT Subroutine

To return to the Keyboard Monitor from a FORTRAN program, use the EXIT subroutine as follows:

```
CALL EXIT
```

7.0 DECTAPE I/O ROUTINES

RTAPE (read tape) and WTAPE (write tape) are the DECtape read and write subprograms for the 8K FORTRAN and 8K SABR systems. For the paper tape FORTRAN system, these subprograms are furnished on one relocatable binary-coded paper tape which must be loaded by the 8K Linking Loader into field 0, where the subprograms occupy one page of core.

RTAPE and WTAPE allow the user to read and write any amount of core-image data onto DECtape in absolute, non-file-structured data blocks. Many such data blocks may be stored on a single tape, and a block may be from 1 to 4096 words in length.

RTAPE and WTAPE may be called with standard, explicit CALL statements in any 8K FORTRAN or SABR program. Each subprogram requires four arguments separated by commas. The arguments are the same for both subprograms and are formatted in the same manner. They specify the following:

1. DECtape unit number (from 0 to 7)

2. Number of the DECtape block at which transfer is to start. The user may direct the DECtape service routine to begin searching for the specified block in the forward direction rather than the usual backward direction by making this argument the two's complement of the block number. For additional information on this and other features, refer to the DECtape Programmer's Reference Manual (DEC-08-SUCO-D).

3. Number of words to be transferred (1<N<4096).

4. Core address at which the transfer is to start.

The general form is:

```
CALL RTAPE (n(1), n(2), n(3), n(4))
```

where n(1) is the DECtape unit number, n(2) is the block number, n(3) is the number of words to be transferred, and n(4) is the starting address.

In 8K FORTRAN, a CALL statement to RTAPE could be written in the following format (arguments are taken as decimal numbers):

```
CALL RTAPE(6,128,388,LOCA)
```

In this example, LOCA may or may not be in common.

As a typical example of the use of RTAPE and WTAPE, assume that you want to store the four arrays A, B, C, and D on a tape with word lengths of 2000, 400, 400, and 20 respectively.

Since PDP-8 DECtape is formatted with 1474 blocks (numbered 0-2701 octal) of 129 words each (for a total of 190,146 words), A, B, C, and D will require 16, 4, 4, and 1 blocks respectively.

The block numbers used by RTAPE and WTAPE should not be confused with the record numbers used by OS/8. An OS/8 record is 256 words--roughly twice the size of a DECtape block. An RTAPE or WTAPE record number is exactly twice the corresponding OS/8 record number. For example, to read the first segment of the OS/8 directory on DECtape #5, the statements:

```
DIMENSION IDIR(258)
CALL RTAPE(5,2,258,IDIR)
```

would read Block 2 (OS/8 Block 1) of DECtape 5.

Each array must be stored beginning at the start of some DECtape block. The user may write these arrays on tape as follows:

```
CALL WTAPE(0,1,2000,A)
CALL WTAPE(0,17,400,B)
CALL WTAPE(0,21,400,C)
CALL WTAPE(0,25,20,D)
```

You may also read or write a large array in sections by specifying only one DECtape block (129 words) at a time. For example, B could be read back into core as follows:

```
CALL RTAPE(0,17,258,B(1))
CALL RTAPE(0,19,129,B(259))
CALL RTAPE(0,20,13,B(388))
```

As shown above, it is possible to read or write less than 129 words starting at the beginning of a DECtape block. It is impossible, however, to read or write starting in the middle of a block. For example, the last 10 words of a DECtape block may not be read without reading the first 119 words as well.

A DECtape read or write is normally initiated with a backward search for the desired block number. To save searching time, you may request RTAPE or WTAPE to start the block number search in the forward direction. This is done by specifying the negative of the block number. Use this method only if the number of the next block to be referenced is at least ten block numbers greater than the last block number used. For example, if you have just read array A and now want array D, you may write:

```
CALL RTAPE(0,1,2000,A)
CALL RTAPE(0,-27,20,D)
```

The following section of a program demonstrates the use of DECtape I/O. Assume that values are already present on the DECtape.

```
      DIMENSION DATA(500)
      .
      .
      .
      NB=0
      SUM=0.
      DO 100 N=1,10
      CALL RTAPE(1,-NB,1500,DATA)
      TEM=0.
      DO 50 K=1,500
50    TEM=TEM+DATA(K)
      SUM=SUM+TEM
100   NB=NB+24
      AMEAN=SUM/5000.
      WRITE (1,110) SUM, AMEAN
      CALL EXIT
110   FORMAT ('SUM=',E15.7' MEAN=',E15.7///)
      END
```

8.0 OS/8 FORTRAN LIBRARY SUBROUTINES

Table 5 contains a summary of the OS/8 FORTRAN library subroutines. This list describes the routines, their functions, and other routines which must be present if the library routines are to be used. The subroutine names listed are the files which comprise OS/8 Source DECtape 3 (available from the Software Distribution Center upon request).

Table 5
FORTRAN II Library Subroutines

Subroutine Name	Entry Points, or Defined External Symbols	Routines That are Pre-requisites	Core Require-ments (Pages)	Function the Routine Performs
IOH	'READ' 'WRITE' 'IOH'	FLOAT UTILTY INTEGR	11	Handles Input and Output Conversion
FLOAT	'FAD' 'FSB' 'FMP' 'FDV' 'STO' 'FLOT' 'FLOAT' 'FIX' 'IFIX' 'IFAD' 'ISTO' 'ABS' 'CHS'	UTILTY	5	Floating Point Arith-metic Package
UTILTY	'OPEN' 'GENIO' 'EXIT' 'ERROR' 'CKIO'	INTEGR	3	FORTRAN De-vice Routines, Error Exit, Normal Exit
POWERS	'IFPOW' 'FFPOW' 'EXP' 'ALOG'	FLOAT UTILTY IPOWRS INTEGR	3	Handles Num-bers to Floating Powers
INTEGR	'IREM' 'IABS' 'DIV' 'MPY' 'IRDSW' 'CLEAR' 'SUBSC'	UTILTY	2	Integer Math Package
TRIG	'SIN' 'COS' 'TAN'	FLOAT	2	Handles Sine, Cosine, and Tangent
ATAN	'ATAN'	FLOAT	2	Handles Arc-tangents
SQRT	'SQRT'	FLOAT UTILTY	1	Handles Square Roots

(continued on next page)

Table 5 (Cont.)
FORTRAN II Library Subroutines

Subroutine Name	Entry Points, or Defined External Symbols	Routines That are Pre-requisites	Core Require-ments (Pages)	Function the Routine Performs
IPOWRS	'IIPOW' 'FIPOW'	FLOAT INTEGR	1	Handles Num-bers to Integer Powers
IOPEN	'IOPEN' 'OOPEN' 'OCLOS' 'CHAIN'	UTILTY	1	OS/8 Device-Independent I/O, and Chaining Routines
RWTAPE	'RTAPE' 'WTAPE'	UTILTY	1	OS/8 Indepen-dent DECtape I/O Routines

9.0 MIXING SABR AND FORTRAN STATEMENTS

An S in column 1 of an input line means that the line has SABR code. This feature is very useful for performing instructions which are undefined in the FORTRAN language. For example:

```
      DIMENSION M(10)
      .
      .
      J=M(1)
      DO 55 K=2,10
      L=M(K)
S     TAD     \L
S     AND     \J
S     DCA     \J
55    CONTINUE
```

This section of code will form the logical AND of M(1) through M(10) in the variable J.

Notice that whenever a FORTRAN variable is used in a SABR statement, the variable name is preceded by a backslash (\). FORTRAN line numbers referenced in SABR statements are also preceded by a backslash for identification purposes. (A backslash is produced by typing a SHIFT/L.)

Information on calling subroutines which are written in SABR assembly language from a FORTRAN program may be found in the description of SABR in this manual.

10.0 SIZE OF A FORTRAN PROGRAM

The maximum size of any FORTRAN program is 36 octal or 30 decimal pages of code.

OS/8 can run FORTRAN programs in 8 to 32K of core. However, no one program or subprogram can be longer than 4K.

You can estimate the size of your program as follows. Take the amount of core available on the system (at least 8K) and from it subtract 4K for the linkage subroutines, external symbol table, and I/O, math, error, and utility subroutines. From the remainder subtract the amount of storage required for data. The remaining space can be used to hold FORTRAN coding, at the rate of 50-70 FORTRAN statements per 1K of core.

One way to have a longer FORTRAN program in core than is usually possible is to divide a FORTRAN program into three chained segments:

Segment 1--inputs data into common storage
Segment 2--FORTRAN program for data processing
Segment 3--does output to desired device(s)

Chaining segments gives two space advantages:

1. The entire program does not have to fit into available core, only the largest segment.

2. If no I/O statements are used in the middle (computational) segment, the I/O conversion routines will not be loaded with that segment. Since these routines occupy over 1100 decimal words, this chaining technique allows the computational segment to be from 50 to 80 statements longer than a similar program containing I/O statements.

When chaining to a subroutine, make certain you have compiled, loaded, and saved a complete runnable main program on the system device. This program is brought into core by the FORTRAN CHAIN subroutine.

11.0 FORTRAN STATEMENT SUMMARY

A summary of the statements available under OS/8 FORTRAN follows.

Table 6
FORTRAN Language Summary

Statement	Definition
Arithmetic Statements	
v=e	v is a variable (scalar or array); e is an expression.
Control Statements	
GOTO n	Transfer control to the statement numbered n.

(continued on next page)

Table 6 (Cont.)
FORTRAN Language Summary

Statement	Definition
Control Statements (Cont.)	
GOTO (n(1),n(2),...,n(i))j	Where n(1)-n(i) are statement numbers and j is a scalar integer variable. This statement transfers control to the j^th member of the series of n(i).
IF(expression)n(1),n(2),n(3)	This statement transfers control to the statement numbered n(1),n(2), or n(3) if the value of the numeric expression is less than, equal to, or greater than zero, respectively. The expression can be simple or complex.
DO n i=m(1),m(2),m(3)	Repeat execution through statement n, beginning with i=m(1), incrementing by m(3), while i is less than or equal to m(2). If m(3) is omitted, it is assumed to be 1. m's and i's cannot be subscripted. m's can be either integer numbers or integer variables; i is an integer variable.
CONTINUE	Dummy statement, used primarily as a target for transfers, particularly the last statement in the range of a DO loop. A DO loop need not end with a CONTINUE statement.
PAUSE PAUSE n	Temporarily suspend execution. The octal equivalent of the decimal number n is displayed in the accumulator. Program execution can be resumed by following the statement with a call to the OPEN subroutine.
STOP	Terminate execution.
END	Terminate compilation; must be the last statement in a program.
Input/Output Statements	
FORMAT (s(1),s(2),...,s(n))	Where S(1)-S(n) are data field specifications, this statement is used with either a READ or WRITE statement.

(continued on next page)

Table 6 (Cont.)
FORTRAN Language Summary

Statement	Definition
Input/Output Statements (Cont.)	
READ (u,f) list	Where u is a device designation (integer constant or integer variable), f is a FORMAT statement number, and list is a list of variables.
WRITE (u,f) list	Where u is a device designation (integer constant or integer variable), f is a format statement number, and list is a list of variables.
Specification Statements	
COMMON v(1),v(2),...,v(n)	Specified variables or arrays are stored in an area available to other programs.
DIMENSION a(1),a(2),...,a(n)	Used to declare variable names to be array names and specify the number and bounds of each one and two dimensional array.
EQUIVALENCE (v(1),v(2), ...,),(v(i),v(i+1),...)	The inclusion of two or more variable or array names in a parenthetical list indicates that the quantities in the list are to share the same memory location and hence have the same value. Subscripts of array variables must be integer constants. Names must not appear in both EQUIVALENCE and COMMON statements.
Subprogram Statements	
FUNCTION v(a(1),a(2),..., a(n))	Declares the program which follows to be a function subprogram. v is the name of the function being defined. v must appear as a scalar variable and be assigned a value during execution of the subprogram.
SUBROUTINE v(a(1),a(2),..., a(n))	Declares the program which follows to be a subroutine subprogram. The arguments in the list(s) are dummy arguments representing the arguments of the subprogram. Dummy arguments must agree in number, order, and type with the arguments used by the calling program.

(continued on next page)

Table 6 (Cont.)
FORTRAN Language Summary

Statement	Definition
Subprogram Statements (Cont.)	
CALL v CALL v (a(1),a(2),...,a(n))	Statement used to transfer control to a subroutine subprogram. v is the subroutine name in the SUBROUTINE statement. The arguments can be of any type, but must agree in number, order, type and array size with the arguments in the SUBROUTINE statement. One or more of the arguments can be used to return results to the calling program. For example: CALL EXIT CALL TEXT (VALUE,123,275) CALL TECK ('MAS',3)
RETURN	Returns control from a subprogram to the calling program. Each subprogram must contain at least one RETURN statement. RETURN cannot be used in the main program.

12.0 FORTRAN ERROR MESSAGES

12.1 Compiler Error Messages

The following OS/8 FORTRAN Compiler error messages are self-explanatory.

```
ARITHMETIC EXPRESSION TOO COMPLEX
EXCESSIVE SUBSCRIPTS
ILLEGAL ARITHMETIC EXPRESSION
ILLEGAL CONSTANT
ILLEGAL CONTINUATION
ILLEGAL EQUIVALENCING
ILLEGAL OR EXCESSIVE DO NESTING
ILLEGAL STATEMENT
ILLEGAL STATEMENT NUMBER
ILLEGAL VARIABLE
MIXED MODE EXPRESSION
SYMBOL TABLE EXCEEDED
SYNTAX ERROR (usually indicates illegal punctuation)
SUBR. OR FUNCT. STMT. NOT FIRST
```

In addition, OS/8 FORTRAN contains the following error messages:

Message	Explanation
COMPILER MALFUNCTION	The meaning of this message has been extended to cover various unlikely Monitor errors.
IO	A device handler has signalled an I/O error.
NO END STATEMENT	The input to the Compiler has been exhausted.
NO ROOM FOR OUTPUT	The file FORTRN.TM cannot fit on the system device.
SABR.SV NOT FOUND	The SABR assembler is not present on the system device.

12.2 Library Error Messages

During execution, the various library programs check for certain errors and print error messages in the form:

XXXX ERROR AT LOC NNNNN

where XXXX is the error code and NNNNN is the location of the error.

Table 7
FORTRAN Library Error Messages

Error Code	Meaning
The following errors are fatal and cause a return to the Keyboard Monitor:	
ALOG	Attempt to compute log of negative number.
IOER	One of the following has occurred: 1. Device-independent input or output attempted without /I or /O options, or user attempted to specify a device requiring a two-page handler for device-independent I/O without using the /H option 2. Bad arguments to IOPEN or OOPEN 3. Transmission error while doing I/O
CHER	File specified as argument to CHAIN not found on system device.
FMT1	Invalid Format statement.

(continued on next page)

Table 7 (Cont.)
FORTRAN Library Error Messages

Error Code	Meaning
The following input errors are fatal unless input is coming from the Teletype, in which case the entire READ statement is tried again:	
FMT2	Illegal character in I format.
FMT3	Illegal character in F or E format.
The following errors do not terminate execution of the user's program.	
DIVZ	Division by zero--very large number is returned.
EXP	Argument to EXP too large--very large number is returned.
OVFL	Floating point overflow--very large number is returned.
FLPW	Negative number raised to floating point power--absolute value taken.
SQRT	Attempt to take square root of negative number--absolute value used.
FIX	Attempt to fix a number >2047; 2047 is returned.

In addition, the error message:

USER ERROR 1 AT XXXX

means that you have tried to reference an entry point of a program which was not loaded, or possibly that you failed to define a subscripted variable in a DIMENSION statement. XXXX has no meaning.

To pinpoint the location of a library program execution error, proceed as follows.

1. Determine, from the storage map, the next lowest numbered location (external symbol) which is the entry point of the program or subprogram containing the error.

2. Subtract, in octal, the entry point location of the program or subprogram containing the error from the location of the error indicated in the error message.

3. From the assembly symbol table, determine the relative address of the external symbol found in step 1 and add that relative address to the result of step 2.

4. The sum of step 3 is the relative address of the error, which can then be compared with the relative addresses of the numbered statements in the program.

Undefined statement numbers are not detected until the assembly phase, at which time a U error message is given. (Refer to the list of SABR error messages.)

INDEX

FLAP/RALF

CONTENTS

CONTENTS (Cont.)

TABLES

FIGURES

1.0 INTRODUCTION

FLAP and RALF are assemblers that translate PDP-8 or PDP-12 processor and floating point processor (FPP) operation codes in a source program into binary codes in two or three passes.

The first pass assigns numeric values to the symbols and places them in the symbol table, the second pass generates the binary coding, and the third pass generates the program listing.

FLAP/RALF is used to assemble programs using the FPP instructions and capabilities. These programs can calculate numeric values as 12-bit integers, 15-bit integers, 24-bit double precision fractions, 3-word floating point values, or 6-word extended-precision floating-point values. Refer to the FPP User's Guide, DEC-12-GQZA-D, for detailed information on the floating point processor and its instruction set.

FLAP can run on an OS/8 System with a floating point processor (FPP) without any other supporting programs. It generates absolute binary output which is legal input to the OS/8 Absolute Loader (ABSLDR). RALF, an extension of FLAP, is part of the OS/8 FORTRAN IV System. It accepts assembly language files and FORTRAN compiler output, and it generates relocatable binary modules that can be loaded by the relocatable loader LOAD (also part of the OS/8 FORTRAN IV System).

The following sections describe the syntax, instruction formats, addressing modes, and pseudo-operators in the assemblers. The special features of RALF involving relocatable assembly are described in Section 14.

2.0 HARDWARE REQUIREMENTS

The minimum hardware configuration for FLAP is a PDP-8 or PDP-12 with a floating point processor (FPP). The minimum hardware configuration for RALF is a PDP-8 or PDP-12 OS/8 System.

3.0 STATEMENT SYNTAX

A source program is a sequence of coding statements in the general format:

```
Label,instruction (space)expression (space)/ comment
```

A physical line of coding may be up to 127-characters long and is terminated by a carriage return. You may use a semicolon in a line of code (except in the comment field) to terminate a logical statement, permitting you to type several statements on a single line. However, a set of logical statements separated by semicolons must not exceed the 127-character limit.

A space is required in a statement:

- after an instruction mnemonic
- before a slash (/) used to indicate a comment
- as an OR operator

Multiple spaces or tabs are equivalent to a single space. These characters are optional after the comma defining a label, after the = sign that sets a value, and before a statement.

3.1 Labels

You can indicate a statement label by preceding that statement with a user-defined symbol followed by a comma. This format assigns the current value of the location counter to the label.

3.2 Instructions

An instruction may be a PDP-8 operation code, an FPP12 operation code, a FLAP pseudo-operator, or a RALF pseudo-operator.

3.3 Expressions

An expression can contain:

- A user-defined symbol (equated symbol or label).
- The symbol ".", which has a value equal to the current location counter.
- A numeric constant.
- Two or more of the above, combined by operators.

FPP and PDP-8 instructions are illegal symbols in expressions. User symbols can be 1 to 6 alphanumeric characters in length and must start with a # or an alphabetic character. Any additional characters are ignored. Thus, the symbols:

```
#100
A
A1234
```

are acceptable, but in the symbol:

```
ASYMBOLMAYBEMORETHAN6CHARACTERS
```

only the first six characters are stored as the symbol name. In this case, all characters after ASYMBO are ignored. You may define up to 500 symbols in an assembly.

All integer expressions are computed in 15-bit 2's complement arithmetic and then truncated if necessary (15 bits for 2-word FPP memory reference instructions and 12 bits for expressions). The following are examples of legal integer (address) expressions:

```
START+1
123
BUFSIZ*2+7600+300
(ADDRES+2
```

The radix pseudo-ops OCTAL and DECIMAL control the interpretation of numbers used in expressions. Decimal numbers larger than 32,767 and octal numbers larger than 77777 will be incorrectly converted and will cause the NE error. (Error messages are listed in Section 14.4.)

3.4 Comments

A comment is a note you add at the end of a line of code, usually to indicate the logical sequence of the program. Type a slash (/), preceded by one or more spaces or tabs, to specify the start of a comment. Comments must not contain angle brackets.

4.0 ARITHMETIC AND LOGICAL OPERATIONS

The FLAP/RALF operators and their functions in combining numbers or symbols to form expressions are as follows.

Operator	Function
+	2's complement addition
-	2's complement subtraction
*	multiplication
/	division
space or tab	inclusive OR used to separate two instructions
!	inclusive OR
"	precedes an ASCII constant; for example, "A has the octal value 301

Expressions are evaluated from left to right. They may not contain floating point constants.

5.0 PDP-8 OPERATION CODES

PDP-8 operation codes are legal defined mnemonics for use with FLAP/RALF. Table 1 lists the mnemonic, octal value, and operation of each PDP-8 operation code. PDP-8 code must be executed by the PDP-8 or PDP-12 processor. Assembler statements using these codes are coded (or executed) in PDP-8 mode.

Table 1
PDP-8 Operation Codes

Mnemonic	Octal	Operation
Memory Reference Instructions		
AND	0000	Logical AND
TAD	1000	2's complement add
ISZ	2000	Increment and skip if zero
DCA	3000	Deposit and clear AC
JMS	4000	Jump to subroutine
JMP	5000	Jump

(continued on next page)

Table 1 (Cont.)
PDP-8 Operation Codes

Mnemonic	Octal	Operation
Group 1 Operate Microinstructions		
NOP	7000	No operation
CLA	7200	Clear AC
CLL	7100	Clear link
CMA	7040	Complement AC
CML	7020	Complement link
RAR	7010	Rotate AC and link right one
RAL	7004	Rotate AC and link left one
RTR	7012	Rotate AC and link right two
RTL	7006	Rotate AC and link left two
IAC	7001	Increment AC
Group 2 Operate Microinstructions		
SMA	7500	Skip on minus AC
SZA	7440	Skip on zero AC
SPA	7510	Skip on positive AC
SNA	7450	Skip on non-zero AC
SNL	7420	Skip on non-zero link
SZL	7430	Skip on zero link
SKP	7410	Skip
OSR	7404	Inclusive OR switch register with AC
HLT	7402	Halt
Combined Microinstructions		
CIA	7401	CMA IAC
LAS	7604	CLA OSR
IOT Microinstructions Keyboard/Reader		
KSF	6031	Skip if keyboard/reader flag=1
KCC	6032	Clear AC and keyboard/reader flag
KRS	6034	Read keyboard/reader buffer
KRB	6036	Clear AC and read keyboard buffer and clear keyboard flag
Teleprinter/Punch		
TSF	6041	Skip if teleprinter/punch flag=1
TCF	6042	Clear teleprinter/punch flag
TPC	6044	Load teleprinter/punch buffer, select and print
TLS	6046	Load teleprinter/punch buffer, select and print, and clear teleprinter/punch flag

(continued on next page)

Table 1 (Cont.)
PDP-8 Operation Codes

Mnemonic	Octal	Operation
Program Interrupt		
ION	6001	Turn interrupt on
IOF	6002	Turn interrupt off
Extended Memory (Type MC8/I)		
CDF	62n1	Change to data field n
CIF	62n2	Change to instruction n
RDF	6214	Read data field into AC
RIF	6224	Read instruction field into AC
RMF	6244	Restore memory field
RIB	6234	Read interrupt

6.0 PDP-8 MODE ADDRESSING

In PDP-8 Mode, addressing is specified by the contents of the Memory Reference Instruction modified by the Data Field and Instruction Field Registers. Direct addressing, specified by bit 3=0, causes reference to the address given in bits 5-11 in page 0 of the current field if bit 4=0, or to the current page if bit 4=1. Indirect addressing, specified by bit 3=1, causes reference to the indirect address contained in the location specified by bits 4-11, used as above. The indirect address for AND,TAD,ISZ, and DCA refers not to the current field but to the field specified in the Data Field Register. The JMP and JMS instructions refer to locations in the field specified in the Instruction Field Register.

The Data Field Register and the Instruction Field Register are originally set through the console switches; however, the registers can be set under program control by means of the CIF and CDF instructions. The CIF instruction sets the Instruction Field Buffer to the specified field. The CDF instruction changes the Data Field Register immediately. Other instructions allow the program to read, save, and restore the Data Field and Instruction Field Registers. Completion of execution of a JMP or JMS instruction sets the Instruction Field Register to the contents of the Instruction Field Buffer. This procedure permits a program to choose a new field, then execute a jump from the current field to an address in the new field.

The character % appended to the end of a memory reference instruction indicates indirect addressing, and the character Z indicates a page 0 reference:

CURRENT PAGE		PAGE ZERO	
DIRECT	INDIRECT	DIRECT	INDIRECT
TAD A	TAD% A	TADZ A	TADZ% A
DCA B	DCA% B	DCAZ B	DCAZ% B

Do not insert spaces between Memory Reference Instructions and either the Z or % character. Also the Z must always precede the % when both are used.

7.0 FPP OPERATION CODES

The Floating Point Processor recognizes three forms of Data Reference Instructions, which are analogous to the Memory Reference Instructions, and three Special Format instruction forms, which are analogous to the Operate Micro-Instructions.

7.1 Data Reference Instructions

Data Reference Instructions cause transfer between memory and the floating point accumulator, a 36-bit register in the FPP. The transfer may be 36 bits of floating point data or 24 bits of double-precision fixed-point fraction data, depending upon where STARTF or STARTD was most recently executed. In the fixed point mode, the last 24 bits of the FAC or memory are used, and the exponent is unchanged.

The descriptions of the instructions contain the following conventional symbols:

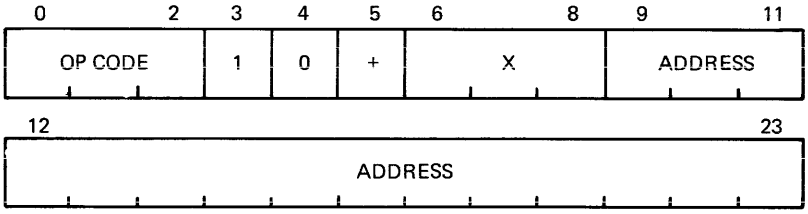
C()	contents of enclosed quantity
FAC	floating accumulator
M	a variable multiplier =2 in Double Precision Mode =3 in Floating Point Mode
X	an indexing variable X=0, do not index $1 \leq X \leq 7$, use specified index register
X0	origin of index registers
Y	address computed
+	an increment bit =0, no incrementing =1, increment before using index
δ(X)	symbol to avoid indexing X=0 δ(X)=0 X=0 δ(X)=1

The op codes, mnemonics, and data functions are:

Op Code	Mnemonic	Data Function
0	FLDA	C(Y)→FAC
1	FADD	C(Y) + C(FAC)→FAC
2	FSUB	C(FAC) - C(Y)→C FAC
3	FDIV	C(FAC)/C(Y)→FAC
4	FMUL	C(FAC * C(Y)→FAC
5	FADDM	C(Y) + C(FAC)→Y
6	FSTA	C(FAC)→Y
7	FMULM	C(FAC) * C(Y)→Y

You can use all eight of the Data Reference Instructions in any of the three forms. The three forms for Data Reference Instructions follow.

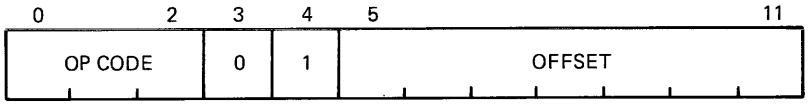
7.1.1 Double-Word Reference Instruction Format



DOUBLE-WORD DATA REFERENCE INSTRUCTIONS

$$Y = C(\text{bits } 9\text{-}23) + M * (C(X + X0) + C(\text{bit } 5)) * \delta(X)$$

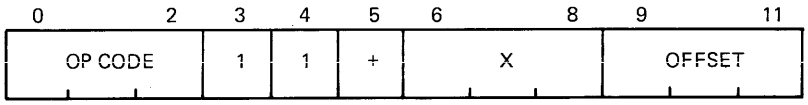
7.1.2 Single-Word Direct Reference Instruction Format



SINGLE-WORD DIRECT REFERENCE

$$Y = C(\text{base register}) + 3 * (\text{offset})$$

7.1.3 Single-Word Indirect Reference Instruction Format



SINGLE-WORD INDIRECT REFERENCE

$$Y = C(\text{bits } 21\text{-}36 \text{ of } C((\text{base register}) + 3 * \text{offset})) + (M) * (C(X + X0) + C(\text{bit } 5)) * \delta(X)$$

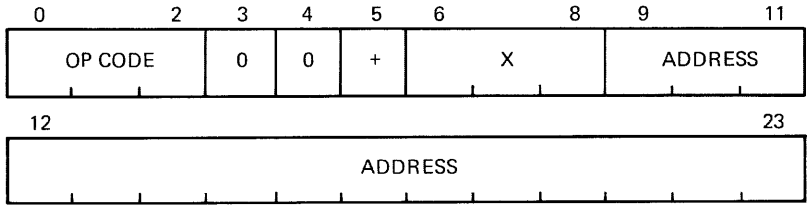
$\delta(X) = 1$ if $X \neq 0$ and 0 if $X = 0$
$M = 2$ if fixed-point mode
$M = 3$ if floating-point mode

7.2 Special Format Instructions

7.2.1 Special Format 1 Instructions: Jump on Count + Trap

Op Code	Mnemonic	Function
2	JXN	If index register X is nonzero, the index register X is incremented if bit 5=1 and a jump is executed to the address contained in bits 9-23.
3 4 5 6 7		The instruction-trap status bit is set and the FPP12 exits, causing a PDP interrupt. The unindexed operand address is dumped into the APT.

The trap instructions with op codes 3 and 4 are assigned a special meaning by RALF. Their mnemonics are TRAP3 and TRAP4 respectively. TRAP3 acts as a JMP to PDP-8 Mode; TRAP4 acts as a JMS to PDP-8 Mode. See the FORTRAN IV Software Support Manual for details.



SPECIAL FORMAT 1

7.2.2 Special Format 2 Instructions

7.2.2.1 Load Index and Add Index

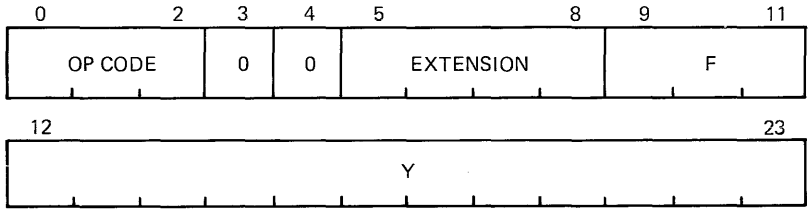
Op Code	Extension	Mnemonic	Function
0	10	LDX	The contents of the index register specified by the bits 9-11 are replaced by the contents of bits 12-23.
0	11	ADDX	The contents of bits 12-23 are added to the index register specified by bits 9-11.

7.2.2.2 **Conditional Jumps** - Jumps are to the location specified by bits 9-23 of the instruction.

Op Code	Extension	Mnemonic	Function
1	0	JEQ	Jump if FAC=0
1	1	JGE	Jump if FAC≥0
1	2	JLE	Jump if FAC≤0
1	3	JA	Jump always
1	4	JNE	Jump if FAC ≠ 0
1	5	JLT	Jump if FAC<0
1	6	JGT	Jump if FAC>0
1	7	JAL	Jump if impossible to fix the floating point number contained in the FAC; that is, if the exponent is greater than 23_{10}.

7.2.2.3 **Pointer Moves**

Op Code	Extension	Mnemonic	Function
1	10	SETX	Set X0 to the address contained in bits 9-23 of the instructions.
1	11	SETB	Set the base register to the address contained in bits 9-23.
1	13	JSR	Jump and save return. Jump to the location specified in bits 9-23, and save the return in bits 21-35 of the first entry of the base page.
1	12	JSA	An unconditional jump to the current address +2 is deposited in the address and address+1, where address is specified by bits 9-23. The FPC is set to address+2.



SPECIAL FORMAT 2

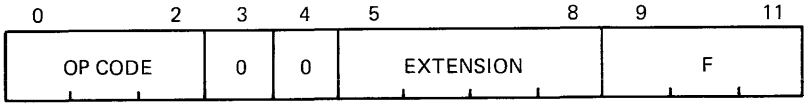
7.2.3 Special Format 3 Instructions

7.2.3.1 Normalize

Op Code	Extension	Mnemonic	Function
0	1	ALN	The mantissa of the FAC is shifted until the FAC exponent equals the contents of the index register specified by bits 9-11. If bits 9-11 are zero, the FAC is aligned so that the exponent = 23_{10}. Setting the exponent = 23_{10} fixes the floating-point number. The JAL instruction tests to see if fixing is possible. In double-precision mode, an arithmetic shift is performed on the FAC fraction. The number of shifts is equal to the absolute value of the contents of the specified index register. The direction of shift depends on the sign of the index register contents. A positive sign indicates a shift toward the least significant bit, while a negative sign indicates a shift toward the most significant bit. The FAC exponent is not altered by the ALN instruction in double-precision mode.
0	2	ATX	The contents of the FAC are fixed and the least significant 12 bits of the mantissa are loaded into the index register specified by bits 9-11. In double-precision mode the least significant 12 bits of the FAC are loaded into the specified index register. The FAC itself is not altered by the FATX instruction.
0	3	XTA	The contents of the index register specified by bits 9-11 are loaded right-justified into the FAC mantissa. The FAC exponent is loaded with 23_{10} and then FAC is normalized. This operation is typically termed floating a 12-bit number. In double-precision mode, the FAC is not normalized.
0	4	NOP	The single-word instruction performs no operation.
0	5-7		These codes are reserved for instruction set expansion and should not be used.
0	12-17	reserved	
1	14-17		

7.2.3.2 Operate

Op Code	Extension	Bits 9-11	Mnemonic	Function
0	0	0	FEXIT	Dump active registers into the APT, reset the FPP RUN flip-flop to the 0 state, and interrupt the PDP-8 processor.
0	0	1	FPAUSE	Wait for synchronizing signal. IOT FFST (6555) will restart the instruction following FPAUSE.
0	0	2	FCLA	Zero the FAC mantissa and exponent.
0	0	3	FNEG	Complement FAC mantissa. This instruction produces the true negative, not the bit-by-bit complement.
0	0	4	FNORM	Normalize the FAC. In double-precision mode FNORM is a NOP.
0	0	5	STARTF	Start floating-point mode.
0	0	6	STARTD	Start double-precision mode.
0	0	7	JAC	Jump to the location specified by the least significant 15 bits of the FAC mantissa.



SPECIAL FORMAT 3

8.0 FPP MODE ADDRESSING

The FLAP/RALF assembler can interact with and effectively use the rather complex addressing scheme of the FPP. This addressing scheme allows the FPP to access a full 32k words of core through 15-bit addresses. It also allows the FPP to access a movable base page through 7-bit addresses. The FPP can also use 2 or 3 bits to specify an index register from a movable set that can modify the address. The FORTRAN compiler makes extensive use of this addressing freedom, particularly in the subroutine calls.

The base page is a block of 128 floating point variables, or 384 12-bit words. The Special Format 2 instruction SETB (see Section 7) gives the FPP the origin of the base page. You can use the pseudo-op BASE to pass the base page origin to the FLAP/RALF assembler. The origin of the base page may be changed as often as necessary. The first 8 locations of the base page serve as a pointer to memory.

The index registers are a block of seven 12-bit words in memory. The Special Format 2 instruction SETX gives the FPP the origin of the index registers. You may change the locations used for the index registers as often as necessary.

The three forms of Data Reference Instructions (see Section 7) compute the address of the data referenced in three different ways. The line of print below the diagram of each instruction shows symbolically how each address is computed. The address computation for the first form (double-word data) begins with the 15-bit address in bits 9-23 of the instruction. If X (bits 6-8) is zero, this is the address used. If X is nonzero, the contents of the specified memory location, X+X0 (where X0 is the beginning of the index registers, set by SETX), is used as an index. If bit 5 of the instruction is equal to one, the index value is incremented by one. The index value remains incremented after the instruction is completed. The resulting index value is multiplied by either two or three, depending upon whether the FPP is in Double Precision Fixed Point Mode (STARTD) or Floating Point Mode (STARTF). This index is then added to the original address (bits 9-23) to form the address used.

The second data reference form (single-word direct) is used to address the locations on the base page. The contents of bits 5-11 of the instruction are multiplied by three and added to the origin of the base page, set by the SETB instruction.

Note tnat the offset on the base page always assumes Floating Point (3-word) variables. It is wise to prevent use of the base page for storage of double-precision fixed-point variables or instructions.

The third form of data reference instruction (single-word indirect) provides an indirect or indexed indirect mode of address. The offset, bits 9-11 of the instruction, are multiplied by three and added to the origin of the base page, to give the address of a 3-word variable. The last 15 bits of this word are used for the address of the data. This address may be modified by the index register exactly the same as in the first form.

The FLAP/RALF Assembler will choose the form of the data reference instruction that is generated. The second form (single-word direct) is used instead of the first form (double-word direct) whenever the data lies on the base page; no indexing is involved. The indirect form is used whenever indirect addressing is called for by a % symbol in the assembler source statement.

9.0 LITERALS

Only FLAP allows literals in PDP-8 code. If you start an expression in a PDP-8 memory reference instruction with a left parenthesis or a square bracket (as explained below), the value after it is taken "literally" by FLAP. Therefore, you do not need to specify an address or label that contains the value. Internally the value of the literal expression is the address of the word generated by FLAP that contains the evaluated expression.

If the expression starts with a left parenthesis, (, then the literal is placed at the end of the current page. If it starts with a left bracket, [, the literal is placed at the end of page 0. Literal tables are built backwards from the end of the page so that the most recently defined literal has the lowest core address.

If the origin is changed to a new page, the previous page's literals are output and the literal table is reset. If the origin is reset to a previous page that contained literals, those literals may be overlaid by any new literals. The previously defined literals will not be available for reference. For this reason, it is best to complete all coding on any non-zero page before moving to another.

If the field is changed, the literals on page 0 of the previous field are output, and the page 0 literal table is reset. For this reason, it is best to complete all coding in any one field before moving to another.

Because locations 0-17 are generally used for interrupts and autoindex registers, only 112(10) (160(8)) literal may be on page 0.

The following examples illustrate the use of literal expressions with memory reference instructions.

TAD (POINTER	generates a literal with the lower 12 bits of the address of POINTER at the end of the current page.
TAD [10	generates a literal containing 0010 at the end of page 0.

The left bracket, [, is typed as a SHIFT/K on an ASR-33.

Literals may not be nested, for example, as in the expression:

```
TAD (TAD [I0
```

10.0 LINKS

Links are generated only by the FLAP assembler. If a PDP-8 memory reference is made to an address that is not on the same page as the instruction, FLAP creates an indirect address linkage on the current page. The address can, therefore, be accessed during the second pass of the Assembler. For example, the coding:

```
                    ORG 200
00200 1777'                 TAD A

00377 0400
                    PAGE
00400 1025          A,      1025
```

is equivalent to

```
                    ORG 200
00200 1777                  TAD I X

                    ORG 377
00377 0400          X,      A

                    PAGE
0400 1025           A,      1025
```

All instructions generating links are flagged in the listing with an apostrophe (') following the generated code. The total number of links is printed at the completion of assembly.

11.0 DATA SPECIFICATION

A logical line of code may consist of only an expression. Such expressions can function as flags, pointers, constants, or symbols. If the expression is larger than 12 bits, it will be truncated to 12 bits.

12.0 PSEUDO-OPERATORS

A pseudo-operator is a defined mnemonic code you include in the source program as a logical line to control some functions of the assembler. Binary code may or may not be generated by a pseudo-op, depending on its function. The FLAP/RALF pseudo-ops and their functions follow.

12.1 ADDR

Generates a two-word address corresponding to the value of the argument.

12.2 BASE n

Places the location of the base page, n, in FLAP/RALF base register for use in calculating single-word addresses. The argument, n, is an expression denoting a 15-bit address. The expression may not contain any symbols that are defined after the BASE pseudo-op occurs. An example of correct sequence follows.

```
ORG 400

A,      F 2.0

B,      F 3.0

        BASE A              /SET ASSEMBLER BASE REGISTER
        SETB A              /SET FPP BASE REGISTER

        FLDA A
```

If no BASE pseudo-op is included, all FPP memory reference instructions will be 2 words. Refer to descriptions on FPP addressing (Section 8) and on referencing memory (Section 13).

12.3 COMMON

Causes the assembler to enter the COMMON section whose name follows the pseudo-op. Subsequent output is placed in the named COMMON section until another section defining pseudo-op is encountered.

12.4 COMMZ

Defines Field 1 8-mode page 0 section. Used to give PDP-8 page 0 section for the Loader.

12.5 DECIMAL

All integers which follow are assumed to be in decimal radix.

12.6 DPCHK

Indicates that the current module requires double precision hardware in order to execute.

12.7 E n

Generates a 6-word extended precision floating point constant with value n. You may write the argument n either as a decimal floating point number or in standard exponential format.

12.8 END

Terminates input. (This pseudo-op is optional; it is never printed on the listing.)

12.9 ENTRY

Defines program entry point. You can use the symbol whose name follows the ENTRY pseudo-op as an external symbol by other programs. Multiple entry points with the same name are accepted by the assembler but cause an error from the loader.

12.10 EQUATE (=)

The symbol to the left of the = is assigned the value of the expression to the right of it.

12.11 EXTERN

Defines the symbol following this pseudo-op to be external to this assembly.

12.12 F n

Generates a 3-word floating point constant with value n. You may write the argument n as a decimal floating point number; for example, 2.0; or in standard exponential format, 2E10. In standard exponential format, 2E10 is equal to 2 x 10^10.

12.13 FIELD1

Defines FIELD1 8-mode section. Used to give field 1 name of section for the Loader.

12.14 IFnnn (Conditional Assembly)

FLAP/RALF have ten conditional pseudo-ops. Four of them require an argument expression:

Pseudo-op	Function
IFZERO n <	assemble if n is zero
IFNZRO n <	assemble if n is not zero
IFPOS n <	assemble if n is positive
IFNEG n <	assemble if n is negative

where n is an integer expression. For each of the above conditional pseudo-ops, the expression n is evaluated and, if it fulfills the conditions of the pseudo-op (for example, n equals zero for IFZERO), the subsequent coding is assembled. If the condition is not met, the subsequent coding is ignored until a matching > is encountered. Assembly is continued after the >.

The fifth and sixth pseudo-ops are used as follows:

IFREF symbol < — assemble if symbol was previously defined or referenced.

IFNDEF symbol < — where symbol may be defined or undefined. When an IFREF statement is encountered, subsequent coding is assembled if the symbol after the pseudo-op has been defined or referenced in a previous statement. The use of a symbol with an IFREF pseudo-op or in a statement that was skipped during assembly because the condition required by a preceding conditional pseudo-op was not met does not constitute a reference to the symbol. If the symbol has not been previously defined or referenced, assembly is continued after the matching > is found.

The seventh through tenth pseudo-ops are:

IFSW n < — assemble the enclosed code if the switch n was set in the input/output file specification to the command decoder, that is, /n or (n).

IFNSW n < — assemble the enclosed code if the switch n was not set.

IFFLAP <	assemble the enclosed code if the assembler is FLAP. This pseudo-op is intended for use in programs which may be assembled either by RALF or by FLAP.
IFRALF <	do not assemble the enclosed code if the assembler is FLAP.

Conditionals may be nested. A possible nested conditional is

```
        IFFLAP < IFREF A < A=263>>
```

Use of some of the conditional assembly pseudo-ops is illustrated in the next example.

```
                          IFPOS -1 <
                    A,    F 0.0
                          >
                          IFNEG -1 <
00200 0000          B,    F 0.0
00201 0000
00202 0000
                          >
                          IFREF A <
                          TAD A
                          >
00203 1200                TAD B
                          >
                          IFREF C <
                          TAD C
                          >
                          IFNDEF D <
                          D=5
                          >

NO ERRORS
2 SYMBOLS, NO LINKS

 B      00200   D          00005
```

12.15 INDEX n

Sets the location of the first FPP index register to n.

12.16 LISTOF

Continues assembly but inhibits further listing. There is no effect on the first two passes or if the listing is currently inhibited. This pseudo-op never appears in the listing.

12.17 LISTON

Ceases to inhibit the listing. There is no effect on the first two passes if the listing is not currently inhibited.

12.18 OCTAL

All integers which follow are assumed to be in octal radix. The digits 8 and 9 are flagged if they occur in octal radix. The radix is initially set to octal by FLAP.

12.19 ORG expr

Assigns the current location counter the value of the lower 15 bits of the address expression expr. The expression should contain only symbols which have previously been defined. For example, to set the origin at location 400 of field 1, the pseudo-op used is ORG 10400.

If the ORG pseudo-op is omitted, an origin of 200 in field 0 is assumed, but the origin setting is not included in the binary output file. For useful results, your program must begin with an ORG pseudo-op.

12.20 PAGE

Sets the current location counter to the beginning of the next core page. This pseudo-op is not in the RALF assembler.

12.21 REPEAT n

Assemble the following line n times.

12.22 S n

Generates a 1-word constant with value n. RALF does not support this pseudo-op.

12.23 SECT

Defines program section, used at the beginning of subprograms to give the name of section for the Loader. For example:

```
        SECT    SUBROU
        JA      START
        BASE     .
  B0,   F       0.
        etc.
```

12.24 SECT8

Defines 8-mode program section. Used at the beginning of 8-mode subprograms.

12.25 TEXT

Enters a string of text. The pseudo-op TEXT is followed by a space or tab, a delimiting character, a string of text, and the same delimiting character, issued in that order. The first printing character after TEXT is the delimiter, and the text string is all the characters that follow it until the next occurrence of the delimiter or a carriage return. The characters space, tab,,, and / cannot be delimiters. For example:

```
TEXT % DATA %
```

causes the word DATA to be printed with the code at assembly time as:

```
00200 0401          TEXT %DATA%
00201 2401
```

12.26 ZBLOCK n

Assembles a block of n words containing 0.

13.0 REFERENCING MEMORY

A PDP-8 computer with an FPP is basically a 32K machine. All of this memory may be referenced through the 15-bit address field provided by the 2-word memory reference instructions. When it is necessary to conserve memory, the base page and the short form (1 word) of the memory reference instructions can be used. Those instructions that have a floating point operand can use this short form:

```
FADD   FDIV   FMUL   FSTA
FADDM  FLDA   FMULM  FSUB
```

The base page is a movable page 0 that you assign. To determine the location referred to by the operand of the single word instruction, multiply the displacement field (address expression) by 3 and add it to the contents of the base register. Thus, when you use the single word form of the instruction, you can reference any location within 128*3 locations of the base register. (Only 128*3 locations can be accessed because the displacement field has only 7 bits.) The location of the base page (via BASE) and the operands (via ORG = etc.) must be defined in the coding before the FPP instruction. Then the short form of the instruction will be executed unless the suffix # is added, forcing the long (2 word) form.

RALF code that includes forward reference to the base page should employ pseudo-ops # and ' as the first character of the symbol; this permits RALF to generate symbols that do not conflict with programmer-generated symbols that are also on the base page. The # pseudo-op can be used following FPP memory reference instructions to indicate use of the 2-word form of the instruction. Likewise, the ' pseudo-op indicates use of the single-word direct form of the instruction.

Consider the following example of the BASE pseudo-op:

```
                    ORG 200
00200 0002          A,      F 2.0
00201 2000
00202 0000
00203 0002          B,      F 3.0
00204 3000
00205 0000
00206 0003          C,      F 5.0
00207 2400
00210 0000
00211 0000          D,      F 0.0
00212 0000
00213 0000

                            BASE 200
                            SETB 200

00214 1110
00215 0200
00216 0200                  FLDA A
00217 1201                  FADD B
00220 4202                  FMUL C
00221 6203                  FSTA D              /D=(A+B)*C
```

This same program can be written with a subroutine:

```
                    ORG 200
00200 0002          A,      F 2.0
00201 2000
00202 0000
00203 0002          B,      F 3.0
00204 3000
00205 0000
00206 0003          C,      F 5.0
00207 2400
00210 0000
00211 0000          D,      F 0.0
00212 0000
00213 0000

00214 1110                  SETB 200
00215 0200
00216 1120                  JSA SUBR
00217 0400
00220 7402                  HLT
                            BASE 0

                    ORG 400

00400 0000          SUBR,   010             /LEAVE 2 WORDS FOR JSA
00401 0000
00402 0200                  FLDA 0          /A
00403 1201                  FADD 3          /B
00404 4202                  FMUL 6          /C
00405 6203                  FSTA 11         /D
00406 1030                  JA SUBR         /--RETURN--
00407 0400
```

This routine performs the same operation as the first one. The values 0, 3, 6, and 11 are used with BASE 0 so that the assembler generates the correct 1-word instructions.

14.0 RALF FEATURES

RALF symbols may be absolute, relocatable, or external. When a relocatable symbol appears in an assembled value, an indicator is placed in the binary output file so that the relocating loader (LOAD) will add the base loading address of the assembled value to arrive at the value to be loaded. If an external symbol appears, the loader will look up the name of the symbol in its symbol table and substitute the value found there for the symbol. The loader symbol table contains all symbols defined by the SECT, SECT8, FIELD1, COMMON, COMMZ and ENTRY pseudo-ops of RALF. Expressions using both absolute and relocatable terms are evaluated as follows (where "op" is one of the set [+-*/&!] and "opl" is one of the set [*/&!]):

Expression	Evaluated
numeric constant	absolute
label	relocatable
absolute op absolute	absolute
relocatable + absolute	relocatable
relocatable - relocatable	absolute
.	relocatable
absolute - relocatable	ERROR
expression opl relocatable	ERROR
relocatable opl expression	ERROR

RALF code is divided into sections; each section is a separately loadable entry within the assembly. These sections are defined via one of the five pseudo-ops: SECT, SECT8, FIELD1, COMMON and COMMZ. Section names are placed in the External Symbol Dictionary (ESD), which is used by the relocating loader to build its symbol table. The pseudo-ops ENTRY and EXTERN allow RALF programs to insert other symbols into the ESD and to refer to these symbols in other RALF programs at load time. Table 3 (Section 14.5) lists the RALF pseudo-ops and their meanings.

14.1 Core Allocation

If you plan to link RALF modules containing PDP-8 mode code, you must be aware of the core allocation algorithm of the loader. Five RALF pseudo-ops may be used to specify a section: SECT, COMMON, SECT8, FIELD1, and COMMZ. These sections are loaded independently by the loader, including those in the same RALF module. SECT is used to begin a section of RALF code that can be loaded into any level and overlay and anywhere in field 1 and above. COMMON is used to begin a section with a given name available to COMMON statements in FORTRAN or other RALF modules. SECT8 is used to begin a section of RALF code that is loaded into level MAIN and is required to begin and end on a page boundary. FIELD1 is used to begin a section subject to all the restrictions of SECT8 and in addition must be loaded into field 1. COMMZ is used to begin a section subject to all the restrictions of FIELD1 and must be loaded into page 0.

The first COMMZ section encountered is forced to begin at location 10000, thus enabling a page 0 in field 1. COMMZ sections of the same name are handled like COMMON sections of the same name (that is, they are combined into one common section). This feature allows 8-mode

code in different modules to share page 0, provided that the modules do not destroy each other's page 0 allocations. In the following example, two modules share page 0, with the first using locations 0-17 and the second using locations 20-37:

```
                        /Module A
          COMMZ SHARE
P1,       1
P2,       2
KSUBA1,   SUBA1
KSUBA2,   SUBA2
          .
          .
          .             /Should not go over
LASTA,    -1            /20 locations

FIELD1    A

          TADZ P1
          JMSZ% KSUBA1
          .
          .
          .             /MODULE B
          COMMZ SHARE
          ORG .+20      /ORG past module A's
                        /Page 0

P3,       3
P4,       4
KSUBB,    SUBB
          .
          .
          .
LASTB     -2
FIELD1    B
          TADZ P3
          .
          .
          .
```

The two COMMZ sections will be put on top of one another; however, because of the ORG .+20 in module B, they will effectively reside back to back. When the image is loaded, the COMMZ sections will look as follows:

```
 LOC        CONTENTS

1 0000         1
  0001         2
     2       SUBA1
     3       SUBA2
     .
     .
     .
1 0017         -1              /LASTA
1 0020          3
    21          4
    22       SUBB
     .
     .
     .
    37         -2              /LASTB
```

If module A is to reference module B's page 0, the procedure is:

```
P3=20
TADZ P3
```

Alternately, a duplicate of the source code for COMMZ SHARE may be included in module B. Modules that are using the same COMMZ section must be aware of how it is divided up. Although COMMZ SHARE takes only 40 locations, the loader allocates a full 200 locations to it. All 8-mode section core allocations are always rounded up so that they terminate on a page boundary. If COMMZ sections of different names exist, they are accepted by the loader and inserted into field 1, but only one COMMZ is the real page 0. In general, it is unwise to have more than 1 COMMZ section name.

If there is more than one COMMZ pseudo-op in a module, they are stacked one behind the other, but there is no way of specifying which one starts at absolute location 0 of field 1. COMMZ sections are allocated by the loader before FIELD1 sections.

If you intend to write 8-mode code that will execute in conjunction with certain 8-mode library routines, note that the layout of PDP-8 FIELD1 #PAGE 0 is:

LOCATION	USE
0-1	Temps for any non-interrupt time routine.
2-13	User locations.
14-157	System locations.
160-177	User locations.

1. Do not define any COMMZ sections other than the system COMMZ which is #PAGE0.

2. If the system page 0 is desired, it will be pulled in from the library if EXTERN #DISP appears in the code.

3. Do not use any part of page 0 reserved for the system.

FIELD1 sections are identical to COMMZ sections in most respects. Memory for FIELD1 sections is allocated after COMMZ sections, however, and FIELD1 sections are combined with FORTRAN COMMON sections of the same name as well as other FIELD1 sections of the same name. The first difference ensures that COMMZ will be allocated page 0 storage even in the presence of FIELD1 sections. The second allows PDP-8 code to be loaded into COMMON, making it possible to load initialization code into data buffers. Two FIELD1 sections with the same name may be combined in the same manner as two COMMZ sections.

The primary purpose of COMMZ is to provide a PDP-8 page 0; the primary purpose of FIELD1 is to ensure that 8-mode code will be loaded into field 1 and that generating CIF CDF instructions in-line is not necessary. SECT8 sections may not be combined in the manner of a COMMON and are not ensured of being placed into field 1.

A section begins when a pseudo-op with its name first appears. A SECT8 section is not combined with another of the same name in another RALF module. However, the second use of the same name in the same module continues a section. For example:

```
SECT8 PARTA
  .
  .
  .
SECT8 PARTB
  .
  .
  .
SECT8 PARTA
```

The second mention of PARTA in the same module continues the source where the first mention of PARTA ended. (Each section has a location counter.)

An 8-mode section does not have to be less than a page in length; however, you should be aware that a SECT8 section that exceeds one page may be loaded across a field boundary and could thereby produce disastrous results at execution time. For this reason, it is generally unwise to cross pages in SECT8 code. This situation will never occur on an 8K configuration. If the total amount of COMMZ and FIELD1 code exceeds 4K, the loader generates an OVER CORE message. The loader generates an MS error for any of the following:

1. A COMMZ section name is identical to some entry point or some non-COMMZ section name.

2. A FIELD1 section name is identical to some entry point or a SECT, SECT8 or COMMZ section name.

3. A SECT8 section name is identical to an entry point or some other section name.

COMMZ sections, like FORTRAN COMMONS, are never entered in the library catalog.

14.2 RALF Programming Notes

The best means of creating RALF modules that can be called from FORTRAN programs is to write a skeleton FORTRAN subroutine. You should write the subroutine so that it can be called with the same "call" statement to be used for the RALF subroutine. This FORTRAN subroutine is then compiled with the RALF output sent to a mass storage file. This file may be modified using EDIT or TECO to create the desired module.

The address pseudo-op (ADDR) which generates a two-word relocatable 15-bit address (that is, JA TAG without use of JA) might prove useful in 8-mode routines. The following example demonstrates a way in which an 8-mode routine in one RALF module calls an 8-mode routine in another module.

```
            EXTERN SUB
            .
            .
            RIF             /Set DF to current
            TAD     ACDF    /IF for return
            DCA     .+1
            0               /CDF X
            TAD     KSUB    /Make a CIF from
            RTL     CLL     /Field bits
            RAL
            TAD     ACIF
            DCA     .+1
            0               /CIF to field
                            /Containing SUB
            JMS%    KSUB+1

KSUB,       ADDR    SUB     /Pseudo-op to
                            /Generate 15 bit
                            /ADDR of subroutine
                            /SUB
ADCF,       CDF
ACIF,       CIF
```

In general the address pseudo-op can be used to supply an 8-mode section with an argument or pointer external to the section.

FPP and 8-mode code may be combined in any RALF section. PDP-8 mode routines must be called in FPP mode by either:

```
          TRAP3 SUB
or        TRAP 4 SUB
```

A TRAP3 SUB causes FRTS to generate a JMP SUB with interrupts on and the FPP hardware (if any) halted. TRAP4 generates a JMS SUB under the same conditions. The return from TRAP4 is:

```
     CDF CIF 0
     JMP% SUB
```

The return from TRAP3 is:

```
     CDF CIF 0
     JMP% RETURN+1

       EXTERN #RETRN
RETURN, ADDR #RETRN
```

It is not possible to call PDP-8 mode subroutines from FORTRAN. A RALF subroutine called from FORTRAN will be entered in FPP-mode; it may branch into PDP-8 mode code using a TRAP3 or TRAP4.

Communication between FPP and 8-mode routines is best done at the FPP level because the FPP mode gives you greater flexibility in both addressing and relocation. The following routine demonstrates how to pass an argument to, and retrieve an argument from, an 8-mode routine:

```
        EXTERN SUB
        EXTERN SUBIN
        EXTERN SUBOUT
        .
        .
        .
        FLDA   X        /Arg for SUB
        FSTA   SUBIN
        TRAP4  SUB      /Call SUB
        FLDA   SUBOUT   /Get result
        FSTA   Y
```

If the 8-mode routine SUB were in the same module as the FPP routine, the EXTERNs would not be necessary. In practice it is common to put in the same section FPP and 8-mode routines that communicate with one another. A number of techniques can be used to pass arguments. For example, an FPP routine could move the index registers to an 8-mode section and pass single precision arguments via ATX.

Because 8-mode routines are commonly used in conjunction with FPP code (generated by the compiler), the 8-mode programmer should be familiar with OS/8 FORTRAN IV subroutine calling conventions. The general code for a subroutine call is a JSR, followed by a JA around a list of arguments, followed by a list of pointers to the arguments. The FPP code for the statement:

```
        CALL SUB (X,Y,Z)
```

would be

```
        EXTERN SUB
        JSR    SUB
        JA     BYARG
        JA     X
        JA     Y
        JA     Z
BYARG,  .
        .
        .
        .
```

The general format of every subroutine obeys the following scheme:

```
        SECT  SUB
        JA    #ST          /Jump to start of
                           /Routine
        TEXT +SUB+         /Needed for
                           /Trace back
RTN,    SETX XSUB          /Reset SUB's index
        SETB BSUB          /And base page
BSUB,   FNOP               /Start of base page
        JA    .
        .
        .
        ORG  BSUB+30       /Restart for SUB
        FNOP:JA RTN
GOBAK,  FNOP:JA.           /Return to
                           /Calling program
```

Location 0000 of the calling routine's base page points to the list of arguments, if any, and may be used by the called subroutine provided that it is not modified. Location 0003 of the calling routine's base page is free for use by the called subroutine. Location 0030 of the calling routine's base page contains the address where execution is to continue upon exit from the subroutine so that a subroutine should not return from a JSR call via location 0 of the calling routine:

```
CORRECT          INCORRECT
FLDA 30          FLDA 0
JAC              JAC
```

This return allows the calling routine to reset its own index registers and base page before continuing in-line execution. General initialization code for a subroutine would be:

```
         SECT     SUB
         JA       #ST
         .
         .
         .
         BASE     0
#ST,     STARTD            /So only 2 words
                           /Will be picked up
         FLDA     30       /Get return JA
         FSTA     GOBAK    /Save it
         FLDA     0        /Get pointer to list
         SETX     XSUB     /Set SUB's XR
         SETB     BSUB     /Set SUB's Base
         BASE     BSUB
         INDEX    XSUB
         FSTA     BSUBX    /Store pointer
                           /Somewhere on Base
         .
         .
         .
         STARTF            /Set F mode before
         JA       GOBAK    /Return
```

The preceding code can be optimized for routines that do not require full generality. The JA #ST around the base page code is a convenience which may be omitted. The three words of text are necessary only for error traceback and may also be omitted. If the subroutine is not going to call any general subroutines, the SETX and SETB instructions at location RTN and the JA RTN at location 0030 are not necessary. If the subroutine does not require a base page, the SETB instruction is not necessary in subroutine initialization; similar remarks apply to index registers. If neither base page nor index registers are modified by the subroutine, the return sequence:

```
FLDA 0
JAC
```

is also legal. In a subroutine call, the JA around the list of arguments is unnecessary when there are no arguments. A RALF listing of a FORTRAN source will provide a good reference of general FPP coding conventions.

The AMOD routine is listed in Figure 1 to illustrate an application of the formal calling sequence. It also includes an error condition check and picks up two arguments. When called from FORTRAN, the code is AMOD (X,Y).

If a PDP-8 mode subroutine is longer than one page and values are to be passed across page boundaries, the address pseudo-op, ADDR, is

required. The format is:

```
     AVAR1, ADDR VAR1
```

```
/
/         A M O D
/         - - - -
/
/SUBROUTINE      AMOD(X,Y)
          SECT    AMOD              /SECTION NAME (REAL NUMBERS)
          ENTRY   MOD               /ENTRY POINT NAME (INTEGERS)
          JA      #AMOD             /JUMP TO START OF ROUTINE
          TEXT    +AMOD +           /FOR ERROR TRACE BACK
AMODXR,   SETX    XRAMOD            /SET INDEX REGISTERS
          SETB    BPAMOD            /ASSIGN BASE PAGE
BPAMOD,   F 0.0                     /BASE PAGE
XRAMOD,   F 0.0                     /INDEX REGS.
AMODX,    F 0.0                     /TEMP STORAGE
          ORG     10*3+BPAMOD       /RETURN SEQUENCE
          FNOP
          JA      AMODXR
          0
AMDRTN,   JA      .                 /EXIT
          EXTERN #ARGER
AMODER,   TRAP4   #ARGER            /PRINT AN ERROR MESSAGE
          FCLA                      /EXIT WITH FAC=0
          JA      AMDRTN
          BASE    0                 /STAY ON CALLER'S BASE PG
/LONG ENOUGH TO GET RETURN ADDRESS
MOD,                                /START OF INTEGER ROUTINE SAME AS
#AMOD,    STARTD                    /START OF REAL NUM. ROUTINE
          FLDA    10*3              /GET RETURN JUMP
          FSTA    AMDRTN            /SAVE IN THIS PROGRAM
          FLDA    0                 /GET POINTER TO PASSED ARG
          SETX    XRAMOD            /ASSIGN MOD'S INDEX REGS
          SETB    BPAMOD            /AND ITS BASE PAGE
          BASE    BPAMOD
          LDX     1,1
          FSTA    BPAMOD
          FLDA%   BPAMOD,1          /ADDR OF X
          FSTA    AMODX
          FLDA%   BPAMOD,1+         /ADDR OF Y
          FSTA    BPAMOD
          STARTF
          FLDA%   BPAMOD            /GET Y
          JEQ     AMODER            /Y=0 IS ERROR
          JGT     .+3
          FNEG                      /ABS VALUE
          FSTA    BPAMOD
          FLDA%   AMODX             /GET X
          JGT     .+5
          FNEG                      /ABS VALUE
          LDX     0,1               /NOTE SIGN
          FSTA    AMODX             /SAV IN A TEMPORARY
          FDIV    BPAMOD            /DIVIDE BY Y
          JAL     AMODER            /TOO BIG.
          ALN     0                 /FIX IT UP NOW.
          FNORM
          FMUL    BPAMOD            /MULTIPLY IT.
          FNEG                      /NEGATE IT.
          FADD    AMODX             /AND ADD IN X.
          JXN     AM,1              /CHECK SIGN
          FNEG
AM,       JA      AMDRTN            /DONE
```

Figure 1 AMOD Routine

This generates a two-word (15 bit) reference to the proper location on another page, here VAR1. For example, to pass a value to VAR1, possible code is:

```
00124   1244    TAD   VAR2      /Value on this page
00125   3757    DCA%  AVAR1+1   /Pass through 12-bit
                                 /location
00156   0000 AVAR1,ADDR VAR1    /Field and
00157   0322                    /location of VAR1
```

Any reference to an absolute address can be effected by the ADDR pseudo-op.

If it is doubtful that the effective address is in the current data field, it is necessary to create a CDF instruction to the proper field. In the above example, suitable code to add to specify the data field is:

```
        TAD  AVAR1          /Get field bits
        RTL                 /Rotate to bits 6-8
        RAL
        TAD  (6201          /Add a CDF
        DCA  .+1            /Deposit in line
        0                   /Execute CDFn
```

If the subroutine includes an off-page reference to another RALF module (for example, in FORLIB), you can address it by using an EXTERN with an ADDR pseudo-op. For example, in the display program, a reference to the non-interrupt task subroutine ONQB is coded as

```
        EXTERN              ONQB
ONQBX   ADDR                ONQB
```

and is called by

```
        JMS%                ONQBX+1
```

No field change instruction is necessary here, because both library modules are defined by field 1 pseudo-op's, and so are both in the same field.

RALF does not recognize LINC instruction or PDP-8 laboratory device instructions. You can include such instructions in the subroutine by defining them with equate statements in the program.

For example, adding the statements:

```
PDP = 2
LINC = 6141
DIS = 140
```

takes care of all instructions for coding the PDP-12 display subroutine.

When you are writing a routine that is going to be longer than a page, it can be useful to have a non-fixed origin in order not to waste core and to facilitate modification of the code. A statement such as

```
        IFPOS .-SECNAM&177-K<ORG
    .-SECNAM&7600+200+SECNAM>
```

will start a new page only if the value [current location less section name] is greater than some K (start of section has a relative value of 0) where K symbol <177 and is the relative location on the current

page before which a new page should be started. The ORG statement includes an AND mask of 7600 to perserve the current page. When it is added to 200 for the next page and the section name, the new origin is set.

When you are calculating directly in a module, the following rules apply to relative and absolute values.

```
relative - relative = absolute
absolute + relative = relative
OR (!), AND (&) and ADD (+) of relative symbols
  generate the RALF error message RE.
```

When you are passing arguments (single precision) from FPP code to PDP code, using the index registers is very efficient. For example,

```
          .
          .
          .
          FLDA%  ARG1    /Get argument in FPP mode
          SETX   MODE8   /Change index registers so XRO is
                         /At MODE8
          ATX    MODE8   /Save argument
          .
          .
          .
          TRAP4  SUB8    /Go to PDP-8 routine
          .
          .
          .
SUB8,     0              /PDP-8 routine
          .
          .
          .
          TAD    MODE8   /Get argument
          .
          .
          .
MODES8,   0              /Index registers set here
          .
          .
          .
```

The source of FORTRAN Library is the best collection available of useful coding techniques in RALF. Working examples include subroutine linkage, 8-mode trap sequences, background task inclusion, interrupt handling, laboratory peripheral interfacing, and mathematical calculation.

14.3 Using the Assembler

To run FLAP/RALF as a standard OS/8 program, type:

```
.R FLAP (or RALF)
*binary,listing< inputl,input2,...
```

Binary is the binary output file (default extension .RI). Listing is the listing output file (default extension .LS). Inputl, input2, etc. are up to 9 source input files, (default extensions .RA). The source files must contain only one FLAP/RALF source module (that is, one END statement).

All error messages and the line that caused the error are printed on the terminal during pass 2, without affecting the binary output file. You may inhibit this output by typing CTRL/O. The error messages are also printed above the error line on the listing. FLAP/RALF error codes are listed in the next section.

You may abort assembly by typing CTRL/C. Each page of a FLAP/RALF listing has a one line header in the form:

```
FLAP (or RALF) V  nn  mo  da,  yrPAGEr
```

where nn is the assembler version number, mo da, yr is the date, and r is the page number.

You may use the /S option, in FLAP, to suppress the listing file and generate only the symbol map on pass 3. If no listing file is specified, the option is ignored. The /T option performs the same function in RALF.

14.4 Error Messages

During pass 2, error messsages are printed at the terminal as they occur. They are followed by the statement in which the error occurred.

During pass 3, error codes are printed in the listing immediately preceding the line in which the error occurred, except the EG message, which is printed after the line. If the line of code includes statements terminated by a semicolon, then the error message for a statement precedes the printing of its octal value on the next line.

A fatal error caused an immediate return to the OS/8 monitor after the message is printed. Table 2 lists the error codes and their meanings.

Table 2
FLAP/RALF Error Codes

Error Code	Meaning
BE	Illegal equate. The symbol had been defined previously.
BI	Illegal index register specification.
BX	Bad expression. Something in the expression is incorrect or the expression is not valid in this context.
DV	An attempt was made in an expression evaluation to divide by zero.
EG	The preceding line contains extra code which could not be used by the assembler.
ES	External symbol error. (RALF only)

(continued on next page)

Table 2 (Cont.)
FLAP/RALF Error Codes

Error Code	Meaning
FL	An error has occurred in the FPP or software floating conversion routines. This could be due to an attempt to convert an excessively large or small number, or an internal error in the assembler occurred.
FP	A syntax error was encountered in a floating point or extended precision constant.
IC	The symbol or expression in a conditional is improperly used, or left angle bracket is missing. The conditional pseudo-op is ignored.
IE	An entry point has not been defined, or is absolute, or is also defined as a common, section, or external. (RALF only)
IL	A literal was used in an instruction which cannot accept one. (FLAP only)
IO	Input/output error (fatal error).
IR	Invalid reference in a PDP-8 instruction.
IX	An index register was specified for an instruction which cannot accept one.
LT	The line is longer than 127 characters. The first 127 characters are assembled and listed.
MD	The tag on the line has been previously encountered at another location or has been used in a context requiring an absolute expression.
NE	Number error. A number out of range was specified or an 8 or 9 occurred in octal radix.
PO	Page overflow. Literals and instructions have been overlapped. (FLAP only)
RE	Relocatability error. A relocatable expression has been used in context requiring an absolute expression. (RALF only)
ST	User symbol table overflow (fatal error).
US	Undefined symbol in an expression.
XS	External symbol table overflow. Control returns to the OS/8 Keyboard Monitor. (RALF only)

14.5 FLAP/RALF Pseudo-operators

Table 3 lists and describes the FLAP/RALF pseudo-ops.

Table 3
FLAP/RALF Pseudo-Operators

Pseudo-op	Meaning
ADDR	Place the 15-bit address of the symbol into two words of core at the current position of the location counter.
BASE expr	Assign base register for 1-word instructions.
COMMON name	Causes the assembler to enter the common section whose name follows the pseudo-op.
COMMZ name	Define name as a special common section restricted to load into page 0 of field 1.
DECIMAL	Set radix for integer conversion to decimal.
E xxx	Generate 6-word extended precision floating point constant.
END	End of input.
ENPUNC	Re-enable binary output (FLAP only).
ENTRY name	Insert name into the ESD as an entry point. The symbol name must be defined as a relocatable symbol in the current assembly.
EXTERN name	Insert name into the ESD as an external reference. The symbol name must not be defined in the current assembly.
F xxx	Generate 3-word floating point constant.
FIELD1 name	Similar to SECT8, but this section is restricted to load into field 1 only.
IFFLAP	Assemble is the assembler if FLAP.
IFNDEF n	Assemble is n is not defined.
IFNEG n	Assemble if n is negative.
IFNSW n	Assemble if switch n was not set in Command Decoder input.

(continued on next page)

Table 3 (Cont.)
FLAP/RALF Pseudo-Operators

Pseudo-op	Meaning
IFNZRO n	Assemble if n is not zero.
IFPOS n	Assemble if n is positive.
IFRALF	Assemble if the assembler if RALF.
IFREF symbol	Assemble if symbol has already been defined or referenced.
IFSW n	Assemble if symbol was set in Command Decoder input.
IFZERO n	Assemble if n is zero.
INDEX n	Assign index register location.
LISTOF	Inhibit program listing.
OCTAL	Set radix for integer conversion to octal.
ORG expr	Set current location counter to lower 15 bits of expr.
PAGE	Set current location counter to the beginning of next core page (FLAP only).
REPEAT n	Repeat next line n times.
S xxx	Generate 1-word constant (FLAP only).
SECT name	Define name as a section and begin that section. Subsequent SECT name commands will resume the section wherever it left off.
SECT8	Similar to SECT, but this section is restricted to load in level MAIN, on a 200(8) word boundary. SECT8 is used to define sections that contain PDP-8 mode code.
TEXT	Assemble the text between delimiters as packed 6-bit ASCII characters.
ZBLOCK n	Assemble n words containing 0.
=	Equate symbol on left of = to value of expression on right.

INDEX

SABR

CONTENTS

CONTENTS (Cont.)

TABLES

1.0 INTRODUCTION

The OS/8 SABR assembler, a modified version of the 8K SABR assembler, is designed to run under the OS/8 Operating System.

You can use the OS/8 SABR assembler as the automatic second pass of the FORTRAN compiler, call it separately to do assemblies of FORTRAN compiled files, or use it as an independent assembler with its own assembly language. In addition, you may use SABR statements in an OS/8 FORTRAN program, expanding the capabilities of the FORTRAN language.

1.1 Calling and Using OS/8 SABR

Unless otherwise specified, OS/8 calls the SABR assembler automatically to assemble the output of a FORTRAN compilation. At other times you can call SABR by typing:

```
R SABR
```

in response to the Keyboard Monitor dot. When the Command Decoder prints an asterisk in the left margin, type the appropriate device assignations, I/O files, and any of the acceptable options.

The line to the Command Decoder consists of 0 to 3 output device and file designations, 1 to 9 input device and file designations, and the desired option(s). The form is:

```
*BINARY,LISTING,MAP<INPUT FILE(S)/OPTION(S)
```

where BINARY represents the binary output, LISTING the listing output, and MAP the Linking Loader loading map input. Unless you indicate alternate extensions, SABR assumes the following extensions:

File Type	Extension
input file	.SB
binary output	.RL
listing output	.LS

If you do not indicate a binary output file, SABR will not generate a binary output. However, if you specify the /L or /G option, SABR will generate a binary file under the assigned name SYS:FORTRL.TM.

1.1.1 **OS/8 SABR Options** - The options you can include in a command string to OS/8 SABR are listed in Table 1.

Table 1
SABR Options

Option	Meaning
/F	Indicates that the input file is an 8K FORTRAN output file.
/G	Calls the Linking Loader, loads the program into core and begins execution. If a binary output file is not specified, then FORTRL.TM is loaded into core and deleted from the file device. If a starting address is not specified (using the options to the Linking Loader), control is sent to the program entry point MAIN (the FORTRAN compiler gives this name automatically to the main program).
/L	Calls the Linking Loader at the end of the assembly and loads the specified binary file. If a binary output file is not specified, then the temporary file FORTRL.TM is loaded into core and deleted from the file device. The Loader then either returns to the Keyboard Monitor with a core image or asks for more input, depending on whether an ALT MODE or RETURN key has terminated the input line.
/N	Outputs the symbol table but not the rest of the listing (applicable only if a listing file is specified).
/S	Omits the symbol table from the listing (applicable only if a listing file is specified).

When you specify the /L or /G option, you can include any options to the Linking Loader (described in Section 13, Linking Loader) in the command string for SABR. You cannot include the /L (Library) option of the Linking Loader, since it would conflict with the SABR /L option.

NOTE

The FORTRAN compiler automatically generates an entry point named MAIN whose address is the beginning of the program. When writing a main program in SABR, specify the entry point MAIN with the entry pseudo-op in order to symbolically specify the starting address to the Linking Loader. (Otherwise you must specify the starting address to the Loader as a five digit address.)

1.1.2 Examples of OS/8 SABR I/O Specification Commands

Example 1:

```
.R SABR
*FORTRN.TM/F/G
```

DSK:FORTRN.TM is assembled as a FORTRAN output file and the relocatable binary is loaded and started at the entry point MAIN.

Example 2:

```
.R SABR
*SYS TEERL,TTY:<TEE/S
```

The input file TEE.SB (or TEE) on DSK: is assembled. The relocatable binary goes to the output file TEERL.RL on SYS:, the listing without a symbol table goes to the terminal.

2.0 THE CHARACTER SET

2.1 Alphabetic

In addition to the letters A through Z, SABR considers the following to be alphabetic:

[	left bracket
]	right bracket
\	back slash
^	up arrow

2.2 Numeric

SABR recognizes the numbers 0 to 9.

2.3 Special Characters

The following printing and non-printing characters are legal:

,	Comma	delimits a symbolic address label
/	Slash	indicates start of a comment
(	Left parenthesis	indicates a literal
"	Quote	precedes an ASCII constant
-	Minus sign	negates a constant
#	Number sign	increases value of preceding symbol by one
	RETURN (carriage return)	terminates a statement
;	Semicolon	terminates an instruction
	LINE FEED	ignored
	FORM FEED	ignored
	SPACE	separates and delimits items on the statement line
	TAB	same as space
	RUBOUT	ignored

All other characters are illegal except when used as ASCII constants following a quote ("), or used in comments or text strings. All legal and illegal characters, used in ways different from the above, cause SABR to print the error message C (Illegal Character).

3.0 STATEMENTS

SABR symbolic programs are a sequence of statements usually prepared on the terminal, on-line, with the aid of the Symbolic Editor program. SABR statements are virtually format free. You terminate each statement by typing the RETURN key. (The Editor automatically provides a line feed.) You can type two or more statements on the same line, using the semicolon as a separator.

You compose a statement line using one or all of the following elements: label, operator, operand and comment -- all separated by spaces or tabs (labels require a following comma). You can identify the types of elements in a statement by the order of appearance in the line and by the separating or delimiting character that follows or precedes the element.

Write statements in the general form:

```
label,  operator  operand  /comment (preceded by slash)
```

SABR generates one or more machine instructions or data words for each source statement.

An input line may be up to 72(10) characters long, including spaces and tabs. Any characters beyond this limit are ignored.

The RETURN key (CR/LF) is both an instruction and a line terminator. You may use the semicolon to terminate an instruction without terminating a line. If, for example, you want to write a sequence of instructions to rotate the contents of the accumulator (AC) and link (L) six places to the right, your instructions might look like this:

```
.
.
RTR
RTR
RTR
.
.
```

You may place all three RTR's on a single line, separating each RTR with a semicolon and terminating the line with the RETURN key. You could then write the preceding sequence of instructions:

```
RTR;RTR;RTR
```

This format is particularly useful when creating a list of data:

```
0200  0020        LIST,    20;50;-30;62
0201  0050
0202  7750
0203  0062
```

You may use null lines to format program listings. A null line is a line containing only a carriage return and possibly spaces or tabs. Such lines appear as blank lines in the program listing.

3.1 Labels

A label is a symbolic name or location tag you created to identify the address of a statement in the program. You can make subsequent references to the statement by referencing the label. If present, the label is written first in a statement and terminated with a comma.

```
0200  0000        SAVE,    0
0201  1200        ABC,     TAD SAVE
```

SAVE and ABC are labels referencing the statements in location 0200 and 0201, respectively.

3.2 Operators

An operator is a symbol or code that indicates an action or operation to be performed, and may be one of the following:

1. A direct or indirect memory reference instruction
2. An operate or IOT microinstruction
3. A pseudo-operator

All SABR operators, microinstructions and memory reference instructions are summarized in the Appendix.

3.3 Operands

An operand represents that part of the statement that is manipulated or operated upon, and may be a numeric constant, a literal or a user-defined address symbol.

In the example last given, SAVE represents an operand.

3.3.1 **Constants** - Constants are data used but not changed by a program. They are of two types: numeric and ASCII. ASCII constants are used only as parameters. You may use numeric constants as parameters or as operand addresses, for example:

```
0200  1412                  TAD I    12
```

SABR treats constant operand addresses as absolute addresses, just as a symbol defined by an ABSYM statement (see Section 5.2, Symbol Definition). References to them are not generally relocatable; therefore, use them only with great care. The primary use of constant operand addresses is to reference locations on page 0. All constant operand addresses are assumed to be in the field into which the Linking Loader loads the program.

Constants may not be added to or subtracted from each other or from symbols.

3.3.1.1 **Numeric Constants** - A numeric constant consists of a single string of from one to four digits. You may precede it with a minus sign (-) to negate the constant. The digit string will be interpreted as either octal or decimal according to the latest permanent mode setting by an OCTAL or DECIM pseudo-operator (explained under Assembly Control). Octal mode is assumed at the beginning of assembly. The digits 8 and 9 must not appear in an octal string.

```
0200  5020        A,      5020
0201  7575                -203
                          DECIM
0202  0120                80
```

3.3.1.2 **ASCII Constants** - You may create eight-bit ASCII values as constants by typing the ASCII character immediately following a double quotation mark ("). You may use a minus sign to negate an alphabetic constant. The minus sign must precede the quotation mark.

```
0200  0273        A,      ";
0201  7477                -"A       /-301
0202  0207                "         /BELL FOLLOWS "
```

The following are illegal as alphabetic constants: carriage return, line feed, form feed and rubout.

3.3.2 **Literals** - A literal is a numeric or ASCII constant preceded by a left parenthesis. The use of literals provides a special and convenient way of generating constant data in a program. The value of the literal will be assembled in a table near the end of the core page on which the instruction referencing it is assembled. The instruction itself will be assembled as an appropriate reference to the location where the numeric value of the literal is assembled. Literals are normally used by TAD and AND instructions, as in the following examples:

```
0200  0376        A,      AND (777
0201  1375                TAD (-50
0202  1374                TAD ("C
      .
      .
      .
0374  0303
0375  7730
0376  0777
```

The numeric conversion mode is initially set to octal, but is controllable with the DECIM and OCTAL pseudo-operators. You can change this mode on a local basis by inserting a D (decimal) or a K (octal) between the left parenthesis and the constant, for example:

```
(D32 becomes 0040 (octal)
(K-32 becomes 7746 (octal)
```

This usage is confined only to the statement in which it is found and does not alter the prevailing conversion mode.

You may also use a literal as a parameter (that is, with no operator). In this case the numeric value of the literal is assembled as usual in the literal table near the end of the core page currently being assembled, and a relocatable pointer to the address of the literal is assembled in the location where the literal parameter appeared.

```
0200  0376 01      A,        (20
      *
      *
      *
0376  0020
```

This feature is intended primarily for use in passing external subroutine arguments with the ARG pseudo-operator, which is explained in greater detail in Section 5.

3.3.3 **Parameters** - A parameter is generally either a numeric constant, a literal or a user-defined address symbol, which is intended to represent data rather than serve as an instruction. It appears as an operand in a statement line containing no operator. (An exception to this is a parameter used in conjunction with the ARG pseudo-operator, explained in Section 6, Subroutines.) In the following example, 200 and -320, M, and PGOADR all represent parameters.

```
0200  0200        ABC,      200;-320;"M
0201  7460
0202  0315
0203  0176        POINTR, PGOADR
```

3.3.4 **Symbols** - Symbols are composed of legal alphanumeric characters and are delimited by a non-alphanumeric character. There are two major types of symbols: permanent, and user-defined.

Permanent Symbols
Permanent symbols are predefined and maintained in SABR's permanent symbol table. They include all of the basic instructions and pseudo-operators in Appendix C. You may use these symbols without prior definition by you.

User-Defined Symbols
A user-defined symbol is a string of from one to six legal alphanumeric characters delimited by a non-alphanumeric character. User-defined symbols must conform to the following rules:

1. The characters must be legal alphanumerics --

   ABCD...XYZ,[]\^ and 0123456789.

2. The first character must be alphabetic.

3. Only the first six characters are meaningful. A symbol such as INTEGER would be interpreted as INTEGE. Since the symbols GEORGE1 and GEORGE2 differ only in the seventh character, they would be treated as the same symbol: GEORGE.

4. A user-defined symbol cannot be the same as any of the pre-defined permanent symbols.

5. A user-defined symbol need be defined only once. Subsequent definitions will be ineffective and will cause SABR to type the error message M (Multiple Definition).

A symbol is defined when it appears as a symbolic address label or when it appears in an ABSYM, COMMN, OPDEF or SKPDF statement (see Section 5, Pseudo-Operators). No more than 64 different user-defined symbols may occur on any one core page.

Equivalent Symbols
When an address label appears alone on a line -- with no instruction or parameter -- the label is assigned the value of the next address assembled.

```
TAG1,
TAG2,    30
TAG3,
```

TAG1 and TAG2 are equivalent symbols in that they are assigned the same value. Therefore, a TAD TAG1 will reference the data at TAG2. TAG3, however, is not equivalent to TAG2. TAG3 would be defined as 1 greater than TAG2.

3.4 Comments

You may add notes to a statement by preceding them with a slash mark. Such comments do not affect assembly or program execution but are useful in interpreting the program listing for later analysis and debugging. Entire lines of comments may be present in the program.

None of the special characters or symbols have significance when they appear in a comment.

```
/THIS IS A COMMENT LINE.
/THIS ALSO. TAD;CALL;#"-2C+=!
A,      TAD SAVE          /SLASH STARTS COMMENT
```

4.0 INCREMENTING OPERANDS

Because SABR is a one-pass assembler and also because it sometimes generates more than one machine instruction for a single user instruction, operand arithmetic is impossible. Statements of the following form are illegal:

```
TAD TAG+3
TAD LIST-LIST2
JMP .+6
```

However, by appending a number sign to an operand you can reference a location exactly one greater than the location of the operand (the next sequential location): TAD LOC# is equivalent to the PAL language statement TAD LOC+1.

```
0200  0020        LOC,     20
0201  0030                 30
0202  1200        START,   TAD LOC    /GET 20
0203  1201                 TAD LOC#   /GET 30
                           PAGE
0400  0200        A,       LOC
0401  0201        B,       LOC#
```

In assembling #-type references, SABR does not attempt to determine if multiple machine code words are generated at the symbolic address referenced.

```
START,  TAD I     LOC       /LOC IS OFF-PAGE
        NOP                 /USER HOPES TO MODIFY
         .
         .
        TAD       (7500     /SMA
        DCA       START#
```

In the preceding example, an attempt was made to change the NOP instruction to an SMA. However, this is not possible because TAD I LOC will be assembled as three machine code words; if START is at 0200, the NOP will be at 0203. The SMA will be inserted at 0201, thus destroying the second word of the TAD I LOC execution.

To avoid this error, you should carefully examine the assembly listing before attempting to modify a program with #-type references. In the previous example the proper sequence is:

```
0202  4067       START,   TAD I LOC
0203  0200 01
0204  1407
0205  7000       VAR,     NOP
0206  1377                TAD (7500
0207  3205                DCA VAR
0377  7500
```

The #-sign feature is intended primarily for manipulating DUMMY variables when picking up arguments from external subroutines and returning from external subroutines (see Section 6.4, Passing Subroutine Arguments).

5.0 PSEUDO-OPERATORS

Table 2 lists the pseudo-operators available in SABR, whether used as a free-standing assembler or in conjunction with the FORTRAN compiler. The pseudo-operators are categorized and explained in the paragraphs following the table.

Table 2
SABR Pseudo-Operators

Mnemonic Code	Operation
ABSYM	Direct absolute symbol definition, used to indicate an absolute core address. For example: ABSYM TEM 177 /PAGE ZERO ADDRESS
ARG	Argument for subroutine call, indicating a value to be transmitted, one value per ARG statement. Used only with CALL. For example: N1, ARG (50 N2, ARG LOCATN

(continued on next page)

Table 2 (Cont.)
SABR Pseudo-Operators

Mnemonic Code	Operation
BLOCK	Reserve storage block; reserves n words of core by placing zeros in them. For example: BLOCK 200 /RESERVE 300 BLOCK 100 /(OCTAL) LOCATIONS
CALL	Call external subroutine. For example: CALL 2,SUBR where 2 is the number of arguments to be passed and SUBR is the subroutine name.
COMMN	Common storage definition, used to name locations in field 1 as externals to be referenced by any program. For example: A, COMMN 20 /20 WORDS IN COMMON
CPAGE	Check if page will hold data, followed by the number of words of code which must be kept together in a unit on a page. That number of words following the CPAGE will be assembled as a unit on the next available core page.
DECIM	Decimal conversion, numeric conversion interprets all numbers input as being decimal numbers.
DUMMY	Dummy argument definition, used in passing arguments to and from subroutines. DUMMY variables are defined in the subprograms which reference them. For example: ENTRY A1 DUMMY X DUMMY Y
EAP	Enter automatic paging mode, restore automatic paging (see LAP).
END	End of program or subprogram.
ENTRY	Define program entry point, used at beginning of subprograms to give name of entry point for the Linking Loader. For example: ENTRY SUBROU SUBROU, BLOCK 2
FORTR	Assemble FORTRAN tape.

(continued on next page)

Table 2 (Cont.)
SABR Pseudo-Operators

Mnemonic Code	Operation
I	Symbolic representation for indirect addressing. For example: DCA I ADD
IF	Conditional assembly, of form: IF NAME, 7 If the symbol NAME has been previously defined, the statement has no effect. If NAME is not defined, the next 7 symbolic instructions are not assembled.
LAP	Leave automatic paging. Assembler is initially set for automatic jumps to the next core page when the current page is full (or upon REORG or PAGE statements). This feature can be suppressed with LAP.
OCTAL	Octal conversion, numeric conversion is originally set to octal and can be changed back to octal after a DECIM pseudo-op has been used.
OPDEF	Define non-skip operator. For example: OPDEF DTRA 6761
PAGE	Terminate current page, begin assembly of succeeding instructions on next core page.
PAUSE	Pause for next tape, designed to allow large source tapes to be broken into several smaller segments. Assembly is continued by pressing the CONT switch.
REORG	Terminate page and reset origin; origin settings are always to the first address of a page. For example: REORG 1000
RETRN	Return from external subroutine, the name of the subroutine being left must be specified. Before the RETRN statement is used, the pointer in the second word of the subprogram entry must be incremented to the point following all arguments in the calling program (after the CALL statement).
SKPDF	Define skip-type operator. For example: SKPDF DTSF 6771

(continued on next page)

Table 2 (Cont.)
SABR Pseudo-Operators

Mnemonic Code	Operation
TEXT	Text string similar to BLOCK, except that the argument is a text string. Characters are stored in six-bit stripped ASCII with a printing character used to delimit the string. For example: TAG, TEXT /123*/ the string would be stored as: 6162 6352 Odd characters are filled with zeros on the right. The floating-point accumulator (in field 1).
ACH	High-order word.
ACM	Middle word.
ACL	Low-order word.

5.1 Assembly Control

END — Every program or subprogram to be assembled must contain the END pseudo-op as its last line. If you do not meet this requirement, an error message (E) is given.

PAUSE — The PAUSE pseudo-op causes assembly to halt and is designed to allow you to break up a large source tape into several smaller segments. To do this, you need only place a PAUSE statement at the end of each section of your source program except the last. Each of these sections of the program is then output as an individual tape. When assembly halts at a PAUSE, remove the source tape just read from the reader and insert the next one. You may then continue assembly by pressing the CONTinue switch.

WARNING

The PAUSE pseudo-op is designed specifically for use at the end of partial tapes and should not be used otherwise.

The reason for this is that the reader routine may have read data from the paper tape into its buffer that is actually beyond the PAUSE statement. Consequently, when you press CONTinue after the PAUSE is found by the line interpreting routine, the entire content of the reader buffer following the PAUSE is destroyed, and the next tape begins reading into a fresh buffer. Thus, if there is any meaningful data on the tape beyond the PAUSE statement, it will be lost.

DECIM — Initially the numeric conversion mode is set for octal conversion. However, if you wish, you may change it to decimal by use of the DECIM pseudo-op.

OCTAL — If the numeric conversion mode has been set to decimal, you may change it back to octal by using the OCTAL pseudo-op.

No matter which conversion mode has been permanently set, it may always be changed locally for literals by use of the (D or (K syntax described earlier, for example:

```
0200  0320       START,  320
                           DECIM
0201  0500               320
0202  0377 01             (K320
0203  1000               512
                           OCTAL
0204  0512               512
0205  0376 01             (D512
0206  0320               320
  .
  .                        END
  .
0376  1000
0377  0320
```

LAP — The assembler is initially set for automatic generation of jumps to the next core page when the page being assembled fills up (Page Escapes), or when PAGE or REORG pseudo-ops are encountered. This feature may be suppressed by use of the LAP (Leave Automatic Paging) pseudo-op.

EAP — If you have previously suppressed the automatic paging feature, you may restore it to operation by using the EAP (Enter Automatic Paging) pseudo-op.

PAGE — The PAGE pseudo-op causes the current core page to be assembled as is. Assembly of succeeding instructions will begin on the next core page. No argument is required.

REORG The REORG pseudo-op is similar to the PAGE pseudo-op, except that you must give a numerical argument specifying the relative location within the subprogram where assembly of succeeding instructions is to begin. You may not give a REORG below 200. A REORG should always be to the first address of a page, it will be converted to the first address of the page it is on.

```
0200  7200      START,  CLA
                      PAGE
0400  7040              CMA
                      REORG 1000
1000  7041              CIA
```

CPAGE The CPAGE pseudo-op followed by a numerical argument N specifies that the following N words of code must be kept together in a single unit and not be split up by page escapes and literal tables. If the N words of code will not fit on the current page of code, the current page is assembled as if a PAGE pseudo-op had been encountered. The N words of code will then be assembled as a unit on the next core page. An example follows.

CPAGE normally specifies a data area. However, if these N words are instructions, for example, a CALL with arguments, you must count the extra machine instructions that SABR must insert.

NOTE

N must be less than or equal to 200 (octal) in nonautomatic paging mode or less than or equal to 176 octal in automatic paging mode.

```
0200  7200      START,  CLA
                      LAP       /INHIBIT PAGE ESCAPE
                      CPAGE 200 /CLOSES THE
0400  0000              NAME1     /CURRENT PAGE
0401  0000              NAME2     /AND ASSEMBLES
                                  /THE NEXT PAGE
                      .
                      .
```

IF Use the conditional pseudo-op, IF, with the following syntax:

```
IF NAME, 7
```

The action of the pseudo-op in this case is to first determine whether the symbol NAME has been previously defined. If NAME is defined, the pseudo-op has no effect. If NAME is not defined, the next seven symbolic instructions (not counting null lines and comment lines) will be treated as comments and not assembled.

```
                    /ABSYM NAME 176
                    IF NAME, 2          /THE NEXT LINE
                            CLL RTL     /TO BE ASSEMBLED
                            RAL         /WILL BE "DCA LOC"
                    /IF THE SLASH BEFORE "ABSYM NAME 176"
                    /IS REMOVED, THE "CLL RTL" AND "RAL"
                    /WILL BE ASSEMBLED.

0200  3201                   DCA LOC
0201  0000            LOC,   0
                             .
                             .
                             .
```

Normally the symbol referenced by an IF statement should be either an undefined symbol or a symbol defined by an ABSYM statement. If this is done, the situation mentioned below cannot occur.

WARNING

In a situation such as the following, a special restriction applies.

```
NAME,   0
.
.
.
IF NAME, 3
```

The restriction is that if the line NAME, 0 happens to occur on the same core page of instructions as the IF statement, then NAME will not have been previously defined when the IF statement is encountered, even though it is before the IF statement. On the first pass (though not in the listing pass) the three lines after the IF statement will not be assembled. The reason for this is that location labels cannot be defined until the page on which they occur is assembled as a unit.

5.2 Symbol Definition

ABSYM — You may name an absolute core address using the ABSYM pseudo-op. This address must be in the same core field as the subprogram in which it is defined. The most common use of this pseudo-op is for naming page zero addresses not used by the operating system. These addresses are listed under Linkage Routine Locations.

OPDEF
SKPDF — Operation codes not already included in the symbol table may be defined by use of the OPDEF or SKPDF pseudo-ops. You must define non-skip instructions with the OPDEF pseudo-op and define skip-type instructions with the SKPDF pseudo-op.

Examples of ABSYM, OPDEF and SKPDF syntax:

```
0177    ABSYM TEM       177   /PAGE 0 ADDRESSES
0010    ABSYM AX        10
6761    OPDEF DTRA      6761  /NON-SKIP INSTR.
6771    SKPDF DTSF      6771  /SKIP-TYPE INSTR.
7540    SKPDF SMZ       7540
```

NOTE

You must make ABSYM, OPDEF and SKPDF definitions before you use them in the program.

COMMN

You use the COMMN pseudo-op to name locations in field 1 as externals so that they may be referenced by any program. If you use any COMMN statements, they must occur at the beginning of the source, before everything else, including the ENTRY statement. Common storage is always in field 1 and is allocated from location 0200 upwards. Since the top page of field 1 is reserved, you may define no more than 3840(10) words of common storage.

A COMMN statement normally takes a symbolic address label, since storage is being allocated. However, you may allocate common storage without an address label.

A COMMN statement always takes a numerical argument that specifies how many words of common storage are to be allocated; however, a 0 argument is allowed. A COMMON statement with 0 argument allocates no common storage; it merely defines the given location symbol at the next free common location.

The syntax of the COMMN statement is as follows:

```
0200      A,      COMMN 20
0220      B,      COMMN 10
0230              COMMN 300
0530      C,      COMMN 0
0530      D,      DOMMN 10
                  ENTRY SUBRUT
```

In this example 20 words of common storage are allocated from 0200 to 0217, and A is defined at location 0200. Then, 10 words are allocated from 0220 to 0227, and B is defined at 0220. Notice that if A is actually a 30 word array, this example equates B(1) with A(21).

The example continues by allocating common storage from 0230 to 0527 with no name being assigned to this block. Then 10 words are allocated from 0530 to 0537 with both C and D being defined at 0530.

5.3 Data Generating

BLOCK

The BLOCK pseudo-op given with a numerical argument N will reserve N words of core by placing zeros in them. This pseudo-op creates binary output, and thus may have a symbolic address label.

Before the N locations are reserved, a check is made to see if enough space is available for them on the current core page. If not, this page is assembled and the N locations are reserved on the next core page. The action here is similar to that of the CPAGE pseudo-op. Similar restrictions on the argument apply.

```
/EXAMPLE OF HOW LARGE BLOCK STORAGE
/MAY BE ACHIEVED WITHIN A SUBPROGRAM AREA

        LAP             /INHIBIT PAGE ESCAPES
        BLOCK 200       /RESERVE 500
        BLOCK 200       /(OCTAL) LOCATIONS
        BLOCK 100
        EAP             /RESUME NORMAL CODING
```

As a special use, if you use the BLOCK pseudo-op with a location label but with no argument or a zero argument, no code zeros are assembled; instead, the symbolic address label is made equivalent to the next relative core location assembled. (This usage is equivalent to using a symbolic address label with no instruction on the same line.)

```
0200  0000      LIST,   BLOCK 3   /ASSEMBLES AS
0201  0000
0202  0000
                                  /THREE ZEROS
                                  /WITH "LIST"
                                  /DEFINED AT THE
                                  /FIRST LOCATION
                NAME1,  BLOCK     /DEFINES NAME1=
                NAME2,  BLOCK 0   /NAME2=NAME3=
                NAME3,            /NAME4
0203  0000      NAME4,  BLOCK 2
0204  0000
```

TEXT

You use the TEXT pseudo-op to obtain packed six-bit ASCII text strings. Its function and use are almost exactly the same as for the BLOCK pseudo-op except that instead of a numerical argument, the argument is a text string. In particular, this pseudo-op makes a check to be sure that the text string will fit on the current page without being interrupted by literals, etc.

You must put the text string argument on the same line as the TEXT pseudo-op. Any printing character may be used to delineate the text string. This character must appear at both the beginning and the end of the string. Carriage return, line feed and form feed are illegal characters within a text string (or as delineators). All characters in the string are stored in simple stripped six-bit form. Thus, a tab character (ASCII

211) will be stored as an 11, which is equivalent to the coding for the letter I. In general, you should not use characters outside the ASCII range of 240-337.

```
0200  2405      TAG,    TEXT      /TEXT EXAMPLE 123*;?/
0201  3024
0202  4005
0203  3001
0204  1520
0205  1405
0206  4061
0207  6263
0210  5273
0211  7700
```

6.0 SUBROUTINES

A subroutine is a subprogram that performs a specific operation and is generally designed so that it can be used more than once or by more than one program. Direction of flow goes from the main, or calling, program to the subroutine, where the action is performed. This is followed by a return back to the address that follows the subroutine call in the main program.

Internal subroutines are those subroutines that can only be called from within a program. You use this type of subroutine extensively in nearly all PDP-8 programs, and you handle it through the use of the JMS, JMS I, and JMP I instructions. An example of an internal subroutine call follows:

```
0200  7300        START,  CLA CLL
0201  1204                TAD N      /GET NUMBER IN AC
0202  4206                JMS TWO    /TRANSFER TO SUB-
                                     /ROUTINE
0203  3205                DCA RESLT  /STORE NUMBER
                                     /(CONTROL RETURNS
                                     /HERE)
0204  0001        N,      1
0205  0000        RESLT,  0

                  /SUBROUTINE
0206  0000        TWO,    0
0207  7104                CLL RAL    /ROTATE LEFT AND
                                     /MULTIPLY BY 2
0210  7430                SZL        /CHECK FOR OVERFLOW
0211  7402        HLT     /STOP IF OVERFLOW
0212  6201 05             JMP I TWO /RETURN TO MAIN
0213  5606
                                     /PROGRAM
                          END
```

The main program picks up a number (N) and jumps to the subroutine (TWO) where N is multiplied by two. A check is made, and if there is no overflow, control returns to the main program through the address stored at the location TWO.

External subroutines are distinguished from internal subroutines in that they may be called by a program that has been compiled, or assembled, without any knowledge of where the subroutine will be located in core memory. Thus, you must load external subroutines with a relocatable linking loader. This makes it possible for you to build a library of frequently used programs and subroutines that you can combine in various configurations. This also eliminates the need to reassemble, or recompile, each individual program when you make a minor change in the system.

A call to an external subroutine can be illustrated using the following FORTRAN programs:

```
          IPARM=5                                    (Calling Program)
          CALL TWO(IPARM)
          WRITE (1,100) IPARM
100       FORMAT (I5)
          END


          SUBROUTINE TWO(IARG)                       (Subroutine)
          IARG=IARG+IARG
          RETURN
          END
```

NOTE

> Exercise care when naming a function or subroutine. It must not have the same name as any of the assembler mnemonics or pseudo-ops or FORTRAN/SABR library functions or subroutines, as errors are likely to result. The symbol table for SABR Assembler is listed in Appendix C, and the library functions are described in the section The Subprogram Library.

Any time a subroutine is called, it must have data to process. This data is contained in parameters in the calling program, which are then passed to the subroutine. The data is picked up by the subroutine where it is referred to as arguments. (The subroutine actually picks up the arguments by a series of TAD I's, and one final TAD I for an integer argument, or by a call to the IFAD subroutine if a floating point argument. This is illustrated in the section entitled SABR Programming Notes.) SABR has special pseudo-operators that facilitate the passing/handling of arguments. Each will be explained in turn.

6.1 CALL and ARG

The CALL pseudo-op is used by the main program to transfer control to the subroutine and is of the form:

```
CALL n,NAME
```

where n represents a one or two-digit number (62(10) maximum) indicating the number of parameters to be passed to the subroutine. NAME (separated from n by a comma) represents the symbolic name of the subroutine entry point.

The Assembler must know the number of parameters that follow the call so that enough room on the current page can be allowed. The CALL pseudo-op and its corresponding parameters must always be coded on the same memory page; that is, there must be no intervening page escapes. (Page format and page escapes are discussed later in the chapter.)

You use the ARG pseudo-op only in conjunction with CALL, and it consists of the symbol ARG, followed by one of the parameters (referred to as arguments in the subroutine) to be passed. You must code one ARG statement for each parameter.

In the previous FORTRAN example, the main program (or it may have been a subroutine) called a subroutine named TWO, and supplied one argument:

```
CALL 1,TWO
ARG IPARM
.
.
.
```

SABR actually assembles the above instructions as follows (you may wish to consult the section concerning the Loader Relocation Codes):

```
0200  0000       IPARM,  BLOCK 1
.
.
.
0206  4033               CALL 1,TWO
0207  0103 06
0210  6201 05            ARG IPARM
0211  0200 01
.
.
.
                 END
```

6.2 ENTRY and RETRN

In the subroutine, the ENTRY statement must occur before the name of the entry point appears as a symbolic address label. The actual entry location must be a two-word reserved space so that both the return address and field can be saved when the routine is called. Execution of the subroutine begins at the first location following the two-word ENTRY block. For example, the TWO subroutine mentioned in the previous example would begin as follows:

```
                         ENTRY TWO
0200  0000       TWO,    BLOCK 2
0201  0000
.
.
.
0227  4040               RETRN TWO
0230  0001 06
                         END
```

When a subroutine is referenced in a CALL statement, the Run-Time Linkage Routine LINK executes the transfer to the subroutine. It assumes that the entry point to the routine is a two-word block. Into the first word of this block it places a CDF instruction which specifies the field of the calling program. In the second word it places the address from which the CALL occurred. (This is analogous to the operation of the JMS instruction.) In the previous example, if the MAIN program had been in field 0, a 6201 would have been deposited in the location at TWO, and a 0210 at TWO #.

The RETRN statement allows you to return to the calling program from the subroutine. You must specify the name of the subroutine being returned from the RETRN statement so that the Return Linkage Routine can determine the action required, and also so that a subroutine may have differently named ENTRY points. (This is analogous to the operation of a JMP I instruction.)

When a subroutine is entered, the second word of the entry name block contains the address of the argument or next instruction that immediately follows the subroutine call in the calling program. It is to this address that control returns.

6.3 Example

Suppose you want to write a long main program, MAIN, which uses two major subroutines, S1 and S2. S1 requires two arguments and S2 one argument. Write MAIN, S1, and S2 as three separate programs in the following manner:

```
        ENTRY MAIN
MAIN,   CLA             /START OF MAIN
        .
        .
        CALL 2,S1
        ARG X
        ARG Y
        CALL 1,S2
        ARG Z
        .
        .
        END


        ENTRY S1
S1,     BLOCK 2
        .
        .
        RETRN S1
        END

        ENTRY S2
S2,     BLOCK 2
        .
        .
        .
        RETRN S2
        END
```

S1 could also contain calls to S2, or S2 calls to S1. Each of these programs is independently assembled with SABR and loaded with the Linking Loader. During the loading process, all of the proper addresses will be saved in tables so that when you begin execution of MAIN, the Run-Time Linkage Routines (see Section 7.3), which were automatically loaded, will be able to execute the proper reference. Thus, MAIN will be able to pass data to and receive it from S1 and S2.

A useful procedure in SABR programming is to provide an ENTRY point named MAIN in the main program at the address where execution is to begin. This assures you that the starting address of the program will appear in the Linking Loader's symbol print-out where it may be easily referenced. If using OS/8, execution will begin at this address automatically, eliminating the need to specify a 5-digit starting address.

6.4 Passing Subroutine Arguments

Use a DUMMY pseudo-op in SABR to define a two word block that contains an argument address.

The format is

```
DUMMY
```

You use indirect instructions to pass arguments to and from subroutines through these DUMMY variables. If a DUMMY variable is referenced indirectly, it causes a CALL to the DUMMY Variable Run-Time Linkage Routine (see Section 7.3, Run-Time Linkage Routines), which assumes that the DUMMY variable is a two-word reserved space where the first word is a 62N1 (CDF N) (N representing the field of the address to be referenced) and that the second word contains a 12-bit address.

As an example, consider the FORTRAN subroutine TWO, shown earlier. You could write this in SABR as follows (you may wish to refer to the section concerning the Subprogram Library):

```
                /CALLED BY:  CALL TWO (IARG)

                         ENTRY TWO     /DEFINE THE
                                       /ENTRY PT. USED
                         DUMMY IARG    /TO PICK UP ARG.
0200  0000       IARG,   BLOCK 2
0201  0000
0202  0000       TWO,    BLOCK 2       /ENTRY POINT
0203  0000
0204  4067               TAD I TWO
0205  0202 01
0206  1407
0207  2203               INC TWO#      /GET ARG ADDRESS
0210  3200               DCA IARG
0211  4067               TAD I TWO
0212  0202 01
0213  1407
0214  2203               INC TWO#
0215  3201               DCA IARG#
0216  4067               TAD I IARG    /GET ARGUMENT
0217  0200 01
0220  1407                             /INTO AC
0221  4067               TAD I IARG    /ADD IT AGAIN
0222  0200 01
```

```
0223  1407
0224  4067              DCA I IARG    /RETURN ARG. TO
0225  0200 01
0226  3407
                                      /CALLING PROGRAM
0227  4040              RETRN TWO
0230  0001 06
                        END
```

A second example may be one in which you have written a FORTRAN program that contains a call to a SABR subroutine ADD:

```
      A=2
      N=3
      CALL ADD(A,N,C)
      WRITE (1,20)C
20    FORMAT (' THE SUM IS',F6.1)
      STOP
      END
```

The FORTRAN program is compiled and the resulting SABR code translates the subroutine call as follows:

```
0223  4033       CALL 3,ADD
0224  0305 06
0225  6201 05    ARG A
0226  0200 01
0227  6201 05    ARG N
0230  0203 01
0231  6201 05    ARG C
0232  0204 01
```

The CALL statement defines 3 parameters -- A, N, and C -- and the subroutine name ADD. The subroutine itself would appear as follows (the DUMMY variables X, K, and Z facilitate the passing of the arguments to and from the subroutine):

```
                  /CALLED BY:  CALL ADD (X,K,Z)
                         ENTRY ADD
                         DUMMY X
                         DUMMY K
                         DUMMY Z
0200  0000       X,      BLOCK 2
0201  0000
0202  0000       K,      BLOCK 2
0203  0000
0204  0000       Z,      BLOCK 2
0205  0000
0206  0200 01    XPNT,   X
0207  0000       PNTR,   0
0210  0000       CNTR,   0
0211  0000       ADD,    BLOCK 2          /ENTRY POINT
0212  0000
0213  1206               TAD XPNT
0214  3207               DCA PNTR
0215  1377               TAD (-6
0216  3210               DCA CNTR
0217  4067       A1,     TAD I ADD
0220  0211 01
0221  1407
0222  2212               INC ADD#
0223  6201 05            DCA I PNTR
0224  3607
0225  2207               INC PNTR
```

```
0226  2210              ISZ CNTR
0227  5217              JMP A1
0230  4067              TAD I K          /GET 2ND ARG
0231  0202 01
0232  1407
0233  4033              CALL 0,FLOT      /CONVERT TO
0234  0002 06
                                         /FLOATING PT.
0235  4033              CALL 1,IFAD      /ADD 1ST ARG
0236  0103 06
0237  6201 05           ARG X
0240  0200 01
0241  4033              CALL 1,ISTO      /RETURN RESULT
0242  0104 06
0243  6201 05           ARG Z
0244  0204 01
0245  4040              RETRN ADD
0246  0001 06
0377  7772
                        END
```

You may use the COMMN pseudo-op to specify variables as externals so that they may be referenced by any program. This pseudo-op has been explained under Symbol Definition; an example of its usage is included here.

```
      0200      C,      COMMN 3          /RESERVES COMMON
                                         /STORAGE
                        ENTRY CSQR       /DEFINES ENTRY PT.
0200  0000      CSQR,   BLOCK 2          /ACTUAL ENTRY POINT
0201  0000
0202  4033              CALL 1,FAD       /GET THE ARGUMENT
0203  0102 06
0204  6211              ARG C
0205  0200
0206  4033              CALL 1,FMP       /MULTIPLY IT
0207  0103 06
0210  6211              ARG C
0211  0200
0212  4033              CALL 1,STO       /REPLACE WITH RESULT
0213  0104 06
0214  6211              ARG C
0215  0200
0216  4040              RETRN CSQR       /RETURN TO CALLING
0217  0001 06
                                         /PROGRAM
                        END
```

This subroutine computes the square of a variable C. C resides in field 1 in common storage where it can be referenced by any calling program through argument passing. The above is equivalent to the FORTRAN subroutine:

```
SUBROUTINE CSQR
COMMON C
C=C*C
RETURN
END
```

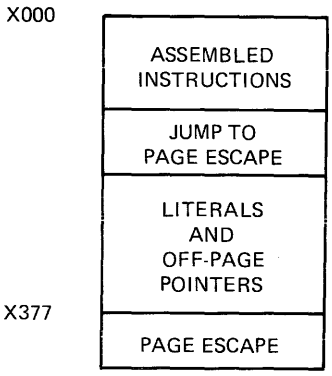
7.0 SABR OPERATING CHARACTERISTICS

7.1 Page-by-Page Assembly

SABR assembles page-by-page rather than one instruction at a time. To accomplish this it builds various tables as it reads instructions. When a full page of instructions has been collected (counting literals, off-page pointers and multiple word instructions) the page is assembled and punched. Several pseudo-operators are available to control page assembly.

7.1.1 Page Format

A normal assembled page of code is formatted as follows:



Literals and off-page pointers are intermingled in the table at the end of the page.

7.1.2 Page Escapes

SABR is normally in automatic paging mode; in this mode, SABR connects each assembled core page to the next by an appropriate jump. This is called a page escape. For the last page of code, SABR leaves the Automatic Paging Mode and issues no page escape. The Leave Automatic Paging (LAP) pseudo-operator turns off the automatic paging mode. EAP (Enter Automatic Paging) turns it back on.

Two types of page escape may be generated. The type generated depends on whether or not the last instruction is a skip. If the last instruction on the page is not a skip, the page escape is as follows:

```
     last instruction (non-skip)
     5377 (JMP to x177)
     literals
     and
     off-page
     pointers
x177/NOP
```

If the last instruction on the page is a skip, the page escape takes four words, as follows:

```
     last instruction (a skip)
     5376 (JMP to x176)
     5377 (JMP to x177)
     literals
     etc.
x176/SKP
x177/SKP
```

7.2 Multiple Word Instructions

Certain instructions in the source program require SABR to assemble more than one machine language instruction (for example, off-page indirect references and indirect references where a data field resetting may be required). In the listing, the source instruction will appear beside the first of the assembled binary words.

A difficulty arises when a multiple word instruction follows a skip instruction. You should be aware that extra instructions are automatically assembled to effect the skip correctly.

7.3 Run-Time Linkage Routines

These routines, which are loaded by the Linking Loader, perform their tasks automatically when certain pseudo-ops or coding sequences are encountered in your program. You need knowledge of them only to better understand the program listing. (Refer to Section 9.1, Loader Relocation Codes.)

There are seven linkage routines:

1.	Change data field to current and skip	CDFSKP
2.	Change data field to 1 (common) and skip	CDZSKP
3.	Off-page indirect reference linkage	OPISUB
4.	Off-bank (common) indirect reference linkage	OBISUB
5.	Dummy variable indirect reference linkage	DUMSUB
6.	Subroutine call linkage	LINK
7.	Subroutine return linkage	RTN

The individual linkage routines function as follows:

1. CDFSKP is called when a direct off-page memory reference follows a skip-type instruction requiring the data field to be reset to the current field.

Program	Assembled Code	Meaning
SZA	7440	
DCA LOC	4045	call CDFSKP
	7410	SKP in case AC = 0 at .-2 execute
	3776	the DCA via a pointer near the end of the page.

2. CDZSKP is called when a direct memory reference is made to a location in common (which is always in Field 1). The action of CDZSKP is the same as that of CDFSKP except that it always executes a CDF 10 instead of a CDF current (see Loader Relocation Codes).

Program	Assembled Code	Meaning
SZA	7440	
DCA CLOC	4051	call CDZSKP
	7410	SKP in case AC = 0 at .-2 execute
	3776	the DCA via a pointer near the end of the page.

3. OPISUB is called when there is an indirect reference to an off-page location.

Program	Assembled Code	Meaning
DCA I PTR	4062	call OPISUB
	0300 01	relative address of PTR
	3407	execute the DCA I via 0007

4. OBISUB is called when there is an indirect reference to a location in common storage. In such a case it is assumed that the location in common which is being indirectly referenced points to some location that is also in common.

Program	Assembled Code	Meaning
DCA I CPTR	4055	call OBISUB
	1000	address of CPTR in Field 1
	3407	execute the DCA I via 0007

5. DUMSUB is called when there is an indirect reference to a DUMMY variable. In such a case, DUMSUB assumes that the DUMMY variable is a two-word vector in which the first word is a 62N1, where N = the field of the address to be referenced, and the second word is the actual address to be referenced.

Program	Assembled Code	Meaning
DCA I DMVR	4067	call DUMSUB
	0300 01	relative address of DMVR
	3407	execute DCA I via pointer in location 0007

6. LINK is called to exeucute the linkage required by a CALL statement in your program. When a CALL statement is used, it is assumed that the entry point of the subprogram is named in the CALL and that this entry point is a two-bit word, free block followed by the executable code of the subprogram. LINK leaves the return address for the CALL in these two words in the same format as a DUMMY variable.

Program	Assembled Code	Meaning
CALL 2, SUBR	4033	call LINK
	0205 06	code word
ARG X	62M1	X resides in field M
	0300 01	relative address of X
	ARG C	6211 C is in common
	1007	absolute address of C

7. RTN is called to execute the linkage by a RETRN statement in the user's program.

Program	Assembled Code	Meaning
RETRN SUBR	4040	call RTN
	0005 06	number of the subrprogram being returned from (SUBR)

7.4 Skip Instructions

In page escapes and multiple word instructions, you must distinguish skip-type instructions from non-skip instructions. For this reason both ISZ and INC are included in the type permanent symbol table. ISZ is considered to be a skip instruction and INC is not. INC should be used to conserve space when you desire to increment a memory word without the possibility of a skip.

The first example below shows the code that is assembled for an indirect reference to an off-page location following an INC instruction. The second example shows the same code following an ISZ instruction.

Example 1:

```
INC POINTR  0220  2376
TAD I LOC2  0221  4062
            0222  0520 01  /OFF PAGE INDIRECT EXECUTION
            0223  1407
```

Example 2:

```
ISZ COUNTR  0220  2376
TAD I LOC2  0221  7410      /SKIP TO EXECUTION
            0222  5226      /JUMP OVER EXECUTION
            0223  4062
            0224  0520 01  /OFF PAGE INDIRECT EXECUTION
            0225  1407
```

You must use a special pseudo-operator, SKPDF, to define skip instructions used in source programs but not included in the permanent symbol table, for example:

```
SKPDF DTSF 6771
```

7.5 Program Addresses

Since each assembly is relocatable, the addresses specified by SABR always begin at 0200, and all other addresses are relative to this address. At loading time, the Linking Loader will properly adjust all addresses. For example, if 0200 and 1000 are the relative addresses of A and B, respectively, and if A is loaded at 2000, then B will be loaded at 2000 + (1000-0200) or 2600.

You must arrange all programs SABR will assemble to fit into one field of memory, not counting page 0 of the field, or the top page (7600 - 7777). If a program is too large to fit into one field, split it into several subprograms.

Explicit CDF or CIF instructions are not needed by SABR programs because of the availability of external subroutine calling and common storage. Explicit CDF or CIF instructions cannot be assembled properly.

7.6 The Symbol Table

Entries in the symbol table are variable in length. A one- or two-character symbol requires three symbol table words. A three- or four-character symbol requires four words, and a five- or six-character symbol, five words. Thus, for long programs it may be to your advantage to use short symbols whenever possible.

The symbol table, not counting permanent symbols, contains 2644(10) words of storage. However, this space must be shared when there are unresolved forward and external references temporarily stored as two-word entries.

If we may assume that a program being assembled never has more than 100(10) of these unresolved references at any one time, this leaves 2464(10) words of storage for symbols. Using an average of four words per symbol, this allows room for 616(10) symbols.

The OS/8 version of SABR has a smaller space for symbol tables, leaving 1364(10) words of storage, or 1620(10) if used as the second pass of FORTRAN II.

Symbol table overflow is a fatal condition that generates the error message S.

Symbols are listed in alphabetic order at the end of assembly pass 1 with their relative addresses beside them. The following flags are added to denote special types of symbols:

ABS	The address referenced by this symbol is absolute.
COM	The address is in common.
OP	The symbol is an operator.

EXT The symbol is an external one and may or may not be defined within this program. If not defined, there is no difficulty; it is defined in another program.

UNDF The symbol is not an external symbol and has not been defined in the program. This is a programmer error. No earlier diagnostic can be given because it is not known that the symbol is undefined until the end of pass 1. A location is reserved for the undefined symbol, but nothing is placed in it.

8.0 THE SUBPROGRAM LIBRARY

The Library is a set of subprograms that may be called by any FORTRAN/SABR program. These subprograms are automatically loaded with the OS/8 FORTRAN/SABR system; in the paper tape system they are provided on two relocatable binary paper tapes with part 1 containing those subprograms used by almost every FORTRAN/SABR program. This allows you to load only those routines which your program makes use of, thus conserving symbol space.

Many of the subprograms reference the Floating-Point Accumulator located at ACH, ACM, ACL (20,21,22 of field 1). The OS/8 Subprogram Library is summarized in the description of FORTRAN II. The organization of the library programs, as they are provided in the paper tape system, is as follows. Descriptions of the programs follow the listing.

```
Part 1.  "IOH"      contains IOH, READ, WRITE
         "FLOAT"    contains FAD, FSB, FMP, FDV, STO,
                             FLOT, FLOAT, FIX, IFIX,
                             IFAD, ISTO, CHS, CLEAR
         "INTEGER"  contains IREM, ABS, IABS, DIV,
                             MPY, IRDSW
         "UTILITY"  contains TTYIN, TTYOUT, HSIN,
                             HSOUT, OPEN, CKIO
         "ERROR"    contains SETERR, CLRERR, ERROR

Part 2.  "SUBSC"    contains SUBSC
         "POWERS"   contains IIPOW, IFPOW, FIPOW,
                             FFPOW, EXP, ALOG
         "SQRT"     contains SQRT
         "TRIG"     contains SIN, COS, TAN
         "ATAN"     contains ATAN
```

8.1 Input/Output

READ is called to initialize the I/O handler before reading data. WRITE is called to initialize the I/O handler before writing data. IOH is called for each item to be read or written. IOH must also be called with a zero argument to terminate an input-output sequence.

Before any of the programs are called, the floating-point accumulator must be set to zero.

```
     CALL      2, READ
     ARG       (n              /n=DEVICE NUMBER
     ARG       fa              /fa=ADDR OF FORMAT
     ooo
     CALL      1, IOH
     ARG       data 1          /data 1=ADDR OF HIGH
                               /ORDER WORD OF
                               /FLOATING POINT
                               /NUMBER
     CALL      1, IOH
     ARG       data 2
     ooo
     ooo
     CALL      1, IOH          /TERMINATES READ
     ARG       0
     ooo
     CALL      2, WRITE        /INITIALIZES WRITE
     ARG       (n
     ARG       fa
```

The following device numbers are currently implemented:

1 (Teletype keyboard/printer)
2 (High-speed reader/punch)
3 (Card reader/line printer)
4 (Assignable device)

8.2 Floating Point Arithmetic

FAD is called to add the argument to the floating-point accumulator.

```
     CALL      1, FAD
     ARG       addres
```

FSB is called to subtract the argument from the floating-point accumulator.

```
     CALL      1, FSB
     ARG       addres
```

FMP is called to multiply the floating-point accumulator by the argument.

```
     CALL      1, FMP
     ARG       addres
```

FDV is called to divide the floating-point accumulator by the argument.

```
     CALL      1, FDV
     ARG       addres
```

CHS is called to change the sign of the floating-point accumulator.

```
     CALL      0, CHS
```

All of the preceding programs leave the result in the floating-point accumulator. The address of the high-order word of the floating-point number is "addres".

STO is called to store the contents of the floating-point accumulator in the argument address. The floating-point accumulator is cleared.

```
     CALL      1, STO
     ARG       storag     /storag=ADDRESS WHERE
                          /RESULT IS TO BE PUT
```

IFAD is called to execute an indirect floating-point add to the floating-point accumulator.

```
     CALL      1, IFAD
     ARG       ptr        /ptr=2 WORD POINTER
                          /TO HIGH ORDER
                          /ADDRESS OF FLOATING
                          /POINT ARGUMENT
```

ISTO is called to execute an indirect floating-point store.

```
     CALL      1, ISTO
     ARG       ptr
```

CLEAR is called to clear the floating-point accumulator. The AC is unchanged.

```
     CALL      0, CLEAR
```

FLOAT and FLOT are called to convert the integer contained in the AC (processor accumulator) to a floating-point number and store it in the floating-point accumulator.

```
                                CALL 1, FLOAT
     CALL      0, FLOT    or
                                ARG  addr
```

IFIX and FIX are called to convert the number in the floating-point accumulator to a 12-bit signed integer and leave the result in the AC.

```
                                CALL 1, IFIX
     CALL      0, FIX     or
                                ARG  addr
```

ABS leaves the absolute value of the floating-point number at "addr" in the floating-point accumulator.

```
     CALL      1, ABS
     ARG       addr
```

8.3 Integer Arithmetic

MPY is called to multiply the integer contained in the AC by the integer contained in "addr." The result is left in the AC.

```
     CALL      1, MPY
     ARG       addr
```

DIV is called to divide the integer contained in the AC by the integer contained in "addr." The result is left in the AC.

```
     CALL      1, DIV
     ARG       addr
```

IREM leaves the remainder from the last executed integer divide in the AC.

```
     CALL      1, IREM
     ARG       0
```

(The argument is ignored.)

IABS leaves the absolute value of the integer contained in "addr" in the AC.

```
     CALL      1, IABS
     ARG       addr
```

IRDSW reads the value set in the console switch register into the AC.

```
     CALL      0, IRDSW
```

8.4 Subscripting

SUBSC, is called to compute the address of a subscripted variable, can be used for doubly or singly subscripted arrays. On entry, the AC should be negative for floating-point variables -- any negative number for singly subscripted variables, and 1's complement of the first dimension for doubly subscripted variables. For doubly subscripted integer variables, the AC must be the first dimension.

The general calling sequence for SUBSC is as follows:

```
TAD (M          /1ST DIMENSION (USED ONLY
                /IF 2 DIMENSIONS)
CMA             /USED ONLY IF ARRAY IS
                /FLOATING POINT
      2,SUBSC   /SINGLE SUBSCRIPT
CALL
      3,SUBSC   /DOUBLE SUBSCRIPT
ARG J           /2ND DIMENSION
ARG I           /1ST DIMENSION
ARG BASE        /BASE ADDRESS OF ARRAY
LOCA            /ADDRESS OF TWO WORD DUMMY
                /ADDRESS LOCATION
```

For example, to load the I,Jth element of a floating-point array whose dimensions are 5 by 7:

```
TAD (5
CMA              /DIMENSIONS ARE 5 BY 7
CALL 3,SUBSC
ARG J            /ADDRESS OF 2ND SUBSCRIPT
ARG I            /ADDRESS OF 1ST SUBSCRIPT
ARG ARRAY        /BASE ADDRESS OF ARRAY
LOC              /MUST BE A DUMMY VARIABLE
CALL 1,IFAD
ARG LOC
```

8.5 Functions

SQRT leaves the square root of the floating-point number at "addr" in the floating-point accumulator.

```
     CALL      1, SQRT
     ARG       addr
```

SIN, COS, TAN leave the specified function of the floating-point argument at "addr" in the floating-point accumulator.

```
     CALL      1, SIN
     ARG       addr
```

ATAN leaves the arctangent of the floating-point number at "addr" in the floating-point accumulator.

```
     CALL      1, ATAN
     ARG       addr
```

ALOG leaves the natural logarithm of the floating-point number at "addr" in the floating-point accumulator.

```
     CALL      1, ALOG
     ARG       addr
```

EXP raises "e" to the power specified by the floating-point number at "addr" and leaves the result in the floating-point accumulator.

```
     CALL      1, EXP
     ARG       addr
```

All of these subprograms require that the floating-point accumulator be set to zero before they are called.

The POWER routines (IIPOW, IFPOW, FIPOW, FFPOW) are called by FORTRAN to implement exponentiation. The first operand is in the AC (floating-point or processor depending on mode), and the address of the second is an argument. The address of the result is in the appropriate AC upon return.

FUNCTION NAME	MODE OF OPERAND 1 (BASE)	MODE OF OPERAND 2 (EXPONENT)	MODE OF RESULT
IIPOW	INTEGER	INTEGER	INTEGER
IFPOW	INTEGER	FLOATING POINT	FLOATING POINT
FIPOW	FLOATING POINT	INTEGER	FLOATING POINT
FFPOW	FLOATING POINT	FLOATING POINT	FLOATING POINT

```
CALL            2, FFPOW
ARG             addr 2          /ADDRESS OF OPERAND 2
```

8.6 Utility Routines

OPEN is called at the beginning of every FORTRAN program to start the high-speed reader/punch and teleprinter, and to initialize the I/O routines for device code 4 if using the OS/8 FORTRAN/SABR system. The form is:

```
CALL 0,OPEN
```

When an error is encountered in a program, the ERROR routine is called. The program passes to the ERROR routine the address of the error message to be printed. The format of the error message is 4 characters in stripped ASCII and packed into 2 words:

```
                        ENTRY ABC
2343  0102      XYZ,    0102;0304
2344  0304
2345  0000      ABC,    BLOCK 2
2346  0000      .
                .
                .
                        CALL 1,ERROR
                        ARG XYZ
```

When control passes to the ERROR routine, the parameters passed are picked up. In the case above, the parameters are as follows:

```
62N1            ARG XYZ
2343
```

where N is the field that XYZ is in, and 2343 is the address of XYZ. The ERROR routine then prints the message at location 2343 plus a 5-digit address which is 2 greater than 2343.

ABCD ERROR AT N2345

Since XYZ is 2 locations before ABC, the address printed will be the address of ABC.

The error message is usually placed just before the entry point of the routine in which the error was detected -- thus the address printed by ERROR will be the address of the entry point. This provides a convenience to you since the entry point will appear in the Loader Map.

CKIO is a subroutine which waits for the TTY flag to be set. It is called by the OS/8 EXIT subroutine to eliminate the possibility of a garbled TTY output. You may use it in FORTRAN for possible expansion with interrupts, and is of the form:

CALL 0,CKIO

The following subroutines -- IOPEN, OOPEN, OCLOSE, CHAIN, EXIT, and GENIO -- are used by the OS/8 FORTRAN/SABR Operating System for device-independent I/O and chaining.

8.7 DECtape I/O Routines

RTAPE and WTAPE (read and write tape) are the DECtape read and write subprograms for the 8K FORTRAN and 8K SABR systems. The subprograms are furnished on one relocatable binary-coded paper tape that must be loaded into field 0 by the 8K Linking Loader, where they occupy one page of core.

RTAPE and WTAPE allow you to read and write any amount of core-image data onto DECtape in absolute, non-file-structured data blocks. Many such data blocks may be stored on a single tape, and a block may be from 1 to 4096 words in length.

RTAPE and WTAPE are subprograms that may be called with standard, explicit CALL statements in any 8K FORTRAN or SABR program. Each subprogram requires four arguments separated by commas. The arguments are the same for both subprograms and are formatted in the same manner. They specify the following:

1. DECtape unit number (from 0 to 7).

2. Number of the DECtape block at which transfer is to start. You may direct the DECtape service routine to begin searching for the specified block in the forward direction, rather than the usual backward direction, by making this argument the two's complement of the block number.

3. Number of words to be transferred (1<N<4096).

4. Core address at which the transfer is to start.

DECtape I/O Routines for the FORTRAN II system are explained in the description of FORTRAN II. In 8K SABR, the CALL statements to RTAPE and WTAPE are written in the following format (arguments may be either octal or decimal numbers):

```
CALL 4,WTAPE     /WOULD BE SAME FOR RTAPE
ARG (6           /DATA UNIT NUMBER
ARG (200         /STARTING BLOCK NUMBER
                 /IN OCTAL
ARG (604         /WORDS TO BE TRANSFERRED
                 /IN OCTAL
ARG LOCB         /CORE ADDRESS, START OF
                 /TRANSFER
```

In these examples, LOCA and LOCB may or may not be in common.

As a typical example of the use of RTAPE and WTAPE, assume that you want to store the four arrays A, B, C, and D on a tape with word lengths of 2000, 400, 400, and 20 respectively. Since PDP-8 DECtape is formatted with 1474 blocks (numbered 0-2701 octal) of 129 words each (for a total of 190,146 words), A, B, C, and D will require 16, 4, 4, and 1 blocks respectively. (Do not confuse the block numbers used by RTAPE and WTAPE with the record numbers used by OS/8. An OS/8 record is 256 words -- roughly twice the size of a DECtape block.)

Each array must start at the beginning of some DECtape block. You may write these arrays on tape as follows:

```
CALL WTAPE (0,1,2000,A)
CALL WTAPE (0,17,400,B)
CALL WTAPE (0,21,400,C)
CALL WTAPE (0,25,20,D)
```

You may also read or write a large array in sections by specifying only one DECtape block (129 words) at a time. For example, B could be read back into core as follows:

```
CALL RTAPE (0,17,258,B(1))
CALL RTAPE (0,19,129,B(259))
CALL RTAPE (0,20,13,B(388))
```

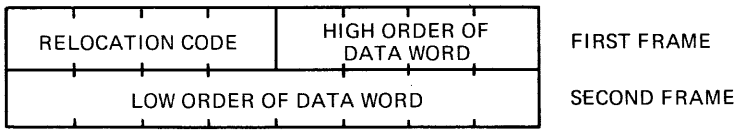
As shown above, it is possible to read or write less than 129 words by starting at the beginning of a DECtape block. It is impossible, however, to read or write starting in the middle of a block. For example, the last 10 words of a DECtape block may not be read without reading the first 119 words as well.

A DECtape read or write is normally initiated with a backward search for the desired block number. To save searching time, you may request RTAPE or WTAPE to start the block number search in the forward direction. This is done by specifying the negative of the block number. This should be used only if the number of the next block to be referenced is at least ten block numbers greater than the last block number used. For example, if you have just read array A and now want array D, you may write:

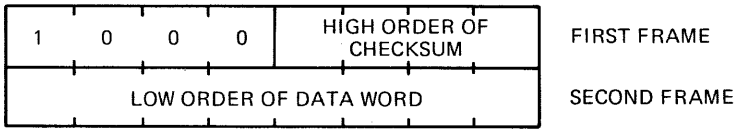
```
CALL RTAPE (0,1,2000,A)
CALL RTAPE (0,-27,20,D)
```

9.0 THE BINARY OUTPUT TAPE

SABR outputs each machine instruction on binary output tape as a 16-bit word contained in two 8-bit frames of paper tape. The first four bits contain the relocation code used by the Linking Loader to determine how to load the data word. The last 12 bits contain the data word itself.



The assembled binary tape is preceded and followed by leader/trailer code (code 200). The checksum is contained in the last two frames of tape before the trailer code. It appears as a normal 16-bit word, as shown below.



All assembled programs have a relative origin of 0200.

9.1 Loader Relocation Codes

The four-bit relocation codes issued by SABR for use by the Linking Loader are explained below. The codes are given in octal.

00	Absolute	Load the data word at the current loading address. No change is required.

```
0205    5277          JMP LOC  /WHERE LOC IS
                       /AT 0077 (OF
                       /CURRENT PAGE)
```

01	Simple Relocation	Add the relocation constant to the word before loading it. (The relocation constant is 200 less than the actual address where the first word of the program is loaded.) Items with this code are always program addresses.

```
0376  0520 01   A,      LOC2
```

In the above example, LOC2 is at relative address 0520. If the first word of the program (relative address 0200) is loaded at 1000, then the actual address of A is 1176, and location 1176 will be loaded with the value 1320, which will be the actual address of LOC2 when loaded.

03	External Symbol Definition*	The data word is the relative address of an entry point. Before entering this definition in the Linkage Tables so that the symbol may be referenced by other programs at run-time, the Linking Loader must add the relocation constant to it. The six frames of paper tape following the two-frame definition are the stripped ASCII code for the symbol.

03	ADDRESS
ADDRESS LOW ORDER	
L	
O	
C	
2	
SPACE	
SPACE	

04	Re-origin*	Change the current loading address to the value specified by the data word plus the relocation constant.
05	CDF Current	The data word is always a 6201 (CDF) instruction, which has been generated automatically by SABR. The code 05 indicates to the Linking Loader that the number of the field currently being loaded into must be inserted in bits 6-8 before loading.

```
0300  6201 05   A,      TAD LOC2
0301  1776                    /WHERE LOC2 IS
                              /OFF PAGE SO
                              /THAT THE TAD
                              /INSTR. MUST BE
                              /INDIRECT
0376  0520 01
```

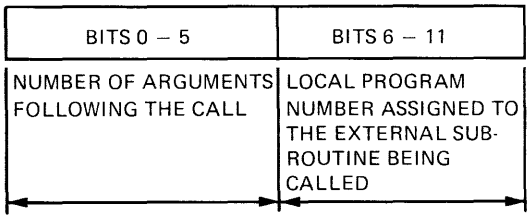
If the program containing this code is being loaded into field 4, relative location 0300 will be loaded with 6241.

Such an instruction is referred to in this document as CDF Current. It is generated automatically by SABR when a direct reference instruction must be assembled as an indirect, and there is the possibility that the current data field setting is different from the field where the indirect reference occurs.

---

* Does not appear in assembly listings

06 Subroutine Linkage Code — The data word is a special constant enabling the Linking Loader to perform the necessary linking for an external subroutine call (compare with CALL Pseudo-op). The structure of the data word is shown below.



Before the 12-bit, two-part code word is loaded into memory, a global external number will be substituted for the local external symbol number in the right half of the data word.

```
0200  4033       CALL 3,SUB
0201  0307 06
                 ARG X
                 ARG Y
                 ARG Z
```

Here, SUB has been assigned the local number 06 during assembly. At loading time this number will be changed to the global number (for example, 23) that is assigned to SUB. In this example, 0323 would actually be loaded at relative address 0201.

10 Leader/Trailer and Checksum* — This code represents normal leader/trailer. The checksum is contained in the last two frames of paper tape preceding the trailer code.

12 High Common* — The data word is the highest location in Field 1 assigned to common storage by the program. This item will occur exactly once in every binary tape and it must be the first word after the leader. If no common storage has been allocated in the program, the data word will be 0177.

* Does not appear in assembly listings

17	Transfer* Vector	Signifies that reference to an external symbol occurs in the assembled program. The 12-bit data word is meaningless. The next six frames contain the ASCII code for the symbol. The Linking Loader uses this definition to create a transfer table, whereby local external symbol numbers assigned during assembly of this particular program can be changed to the global external symbol number when several programs are being loaded.

10.0 SAMPLE ASSEMBLY LISTINGS

The following examples illustrate many of the features and formats of the SABR Assembler.

When a multiple-word instruction occurs, the actual instruction line is typed beside the first instruction.

```
0650  6201 05   LOC2,   JMP NAME  /OFF PAGE
0651  5774
0652  7106              CLL RTL;RTL;RTL
0653  7006
0654  7006
```

When an erroneous instruction occurs, the error flag appears in the address field. The instruction is not assembled.

```
0700  7200      N2,     CLA
   I                    CLL SKP
0701  7402              HLT
```

The page escape and literal and off-page pointer table are typed with nothing except the correct address, value and loader code.

```
0770  7006      N3,     RTL
0771  7500              SMA
0772  5376
0773  5377
0774  0200 01
0775  0020
0776  7410                    /SKP TO 1ST LOC.-
                              /NEXT PAGE (AC IS
                              /NOT MINUS)
0777  7410                    /SKP TO 2ND LOC.-
                              /NEXT PAGE (AC IS
                              /MINUS)
```

Locations 0772, 0773, 0776 and 0777 make up the page escape since the last instruction is a skip instruction (SMA). Refer to Section 7.1.2, Page Escapes.

The following program has been assembled and listed. It cannot be run without first debugging and editing it.

---

* Does not appear in assembly listings

During the first pass, SABR outputs the binary tape and prints error messages as they occur. In this case, none of the errors are fatal, and assembly continues. The symbol table is printed, and undefined symbols, external symbols, or any other special types of symbols that cannot be determined until the end of the pass are flagged in the symbol table.

The optional second pass of the Assembler produces a listing. The 4-digit first column contains the octal address, while the second column contains the octal code for each line of instructions. Errors are also printed during the listing pass at the line in which they occur. Error codes are described in Section 12.

The reader is also referred to Section 16, Demonstration Program Using Library Routines.

```
C AT PUNCH +0003


COUNT   0302
DECIMA  0000UNDF
LT      0264
MAIN    0000EXT
MESG    0243
ORG     0303
PTAPE   0201EXT
PUNCH   0274
REF     0177ABS
RPT     0267
START   0205
TYPE    0000EXT


                    /PROGRAM TO PUNCH RIM FORMAT PAPER TAPES

      6026                  OPDEF PLS 6026  /DEFINE HI SPEED
      6021                  SKPDF PSF 6021  /IOTS
      0177                  ABSYM REF 177
                            ENTRY MAIN
0200  0000                  DECIMAL
                            LAP
0201  0000        PTAPE,    BLOCK 2         /PUNCH LEADER
0202  0000
                                            /TAPE (200 CODE)
0203  1377                  TAD (-32        /32 LOCATIONS
0204  3302                  DCA COUNT
                            OCTAL
0205  1303        START,    TAD ORG
0206  7132                  CLL CML RTR;RTR;RTR
0207  7012
0210  7012
0211  0376                  AND (177
0212  4274                  JMS PUNCH       /PUNCH LEADING
0213  1303                  TAD ORG         /DIGITS OF ADDRESS
0214  0376                  AND (177        /PUNCH SECOND
0215  4274                  JMS PUNCH       /DIGITS OF ADDRESS
0216  1703                  TAD I ORG       /NOW PUNCH CONTENTS
0217  7112                  CLL RTR;RTR;RTR /OF THAT LOCATION
0220  7012
0221  7012
0222  0375                  AND (77
0223  4274                  JMS PUNCH
0224  1703                  TAD I ORG       /GET SECOND DIGITS
0225  0375                  AND (77         /OF THAT LOCATION
```

```
0226  4274              JMS PUNCH
0227  2303              INC ORG           /POINT TO NEXT
                                          /CORE LOCATION
0230  2302              ISZ COUNT         /DONE YET?
0231  5205              JMP START         /NO
0232  4033              CALL 1,TYPE       /YES, TYPE MESSAGE
0233  0102 06
0234  6201 05           ARG MESG
0235  0243 01
0236  4264              JMS LT            /ENDING 200 CODE
0237  7404              OSR               /GET NEW ADDRESS
0240  3303              DCA ORG           /FROM SWITCH REGISTER
                                          /PUT IT IN ORG
0241  7402              HLT               /PAUSE
0242  5774              JMP MAIN          /PUNCH NEW TAPE

0243  2401      MESG,   TEXT "TAPE PUNCHED. ENTER ORIGIN' & CONT."
0244  2005
0245  4020
0246  2516
0247  0310
0250  0504
0251  5640
0252  0516
0253  2405
0254  2240
0255  1722
0256  1107
0257  1116
0260  4046
0261  4003
0262  1716
0263  2456

0264  0000      LT,     0
                        OCTAL
0265  1373              TAD (-40
0266  3302              DCA COUNT         /32 FRAMES OF
0267  1372      RPT,    TAD (200          /LEADER/TRAILER
0270  4274              JMS PUNCH         /PUNCH IT
0271  2302              ISZ COUNT         /DONE?
0272  5267              JMP RPT           /NO
0273  5664              JMP I LT          /RETURN
0274  0000      PUNCH,  0
0275  6026              PLS               /PUNCH
0276  6021              PSF               /WAIT FOR FLAG
    C                   JMP .-1
0277  4045              JMP I PUNCH       /EXIT
0300  7410
0301  5674

0302  0000      COUNT,  0
0303  7300      ORG,    7300
0304  4040              RETRN PTAPE
0305  0003 06
0372  0200
0373  7740
0375  0077
0376  0177
0377  7746
                        END
```

11.0 SABR PROGRAMMING NOTES

11.1 Optimizing SABR Code

Generally two types of programmers will use the SABR Assembler: those who like the convenience of a page-boundary-independent code and need not be concerned with program size, and those who need a relocatable assembler but are still location conscious. These optimizing hints are directed to the latter user.

One way to circumvent the cost of non-paged code is to make use of the LAP (Leave Automatic Paging) pseudo-op and the PAGE pseudo-op to force paging where needed. This saves 2 to 4 instructions per page by elimination of the page escape. In addition, the fact that the program must be properly segmented may save a considerable amount.

Extra core may be reduced by eliminating the CDF instructions which SABR inserts into a program. This is done by using "fake indirects". Define the following op codes:

```
OPDEF ANDI 0400
OPDEF TADI 1400
OPDEF ISZI 2400
OPDEF DCAI 3400
```

These codes correspond to the PDP-8 memory reference instructions but they include an indirect bit. The difference can best be illustrated by an example. If X is off-page, the sequence:

```
LABEL,  SZA
        DCA X
```

is assembled by SABR into:

```
LABEL,  SZA
        JMS 45
        SKP
        DCA I (X)
```

or four instructions and one literal.

The sequence:

```
PX,     X
        .
        .
        .
LABEL,  SZA
        DCAI PX
```

assembles into three instructions for a saving of 40 percent. However, you must be sure that the data field will be correct when the code at LABEL is encountered. Also note that SABR assumes that the Data Field is equal to the Instruction Field after a JMS instruction, so subroutine returns should not use the JMP I op code.

The standard method to fetch a scalar integer argument of a subroutine in SABR is:

```
                        DUMMY X
0200  0000      IARG,   0
0201  0000      X,      BLOCK 2
0202  0000
0203  0000      SUBR,   BLOCK 2
0204  0000
0205  4067              TAD I SUBR
0206  0203 01
0207  1407
0210  3201              DCA X
0211  2204              INC SUBR#
0212  4067              TAD I SUBR
0213  0203 01
0214  1407
0215  3202              DCA X#
0216  2204              INC SUBR#
0217  4067              TAD I X
0220  0201 01
0221  1407
0222  3200              DCA IARG
                        .
                        .
                        .
```

This is the method the FORTRAN compiler uses, and although it is standard, it is also the slowest. This code requires 19 words of core and takes several hundred microseconds to execute.

The fastest way to pick up arguments within a SABR-coded external subroutine is as follows (this method takes approximately one fifth of the time of the previous method and four locations fewer):

```
0200  0000      IARG,   0
0201  0000      SUBR,   BLOCK 2
0202  0000
0203  1201              TAD SUBR
0204  3205              DCA X1
0205  7402      X1,     HLT             /REPLACED
                                        /BY CDF
0206  1602              TADI SUBR#
0207  3214              DCA X2
0210  2202              INC SUBR#
0211  1602              TADI SUBR#
0212  3200              DCA IARG
0213  2202              INC SUBR#
0214  7402      X2,     HLT             /REPLACED
                                        /BY CDF
0215  1600              TADI IARG
0216  3200              DCA IARG
                        .
                        .
```

To pick up multiple arguments, you can make the locations from X1 to X2+1 inclusive into a subroutine.

11.2 Calling the OS/8 USR and Device Handlers

One important point to remember is that any code which calls the USR must not reside in locations 10000 to 11777. Therefore, any SABR routine which calls the USR must be loaded into a field other than field 1 or above location 2000 in field 1. To call the USR from SABR use the sequence:

```
CPAGE N          /N=7+(# OF ARGUMENTS)
6212             /CIF 10
JMS 7700         /OR 200 IF USR IN CORE
REQUEST
ARGUMENTS        /OPTIONAL DEPENDING ON REQUEST
ERROR RETURN     /OPTIONAL DEPENDING ON REQUEST
```

To call a device handler from SABR, use the sequence:

```
        CPAGE 12         /10 IF "HAND" IN PAGE 0
        6202             /CIF 0
        JMS I HAND       /DO NOT USE JMSI
        FUNCT
        ADDR
        BLOCK
        ERROR RETURN
        SKP
HAND,   0                /"HAND" MUST BE ON SAME PAGE
                         /AS CALL, OR IN PAGE 0
```

12.0 SABR ERRORS

In case of error, SABR prints error codes in the address field of the instruction line. Table 3 lists SABR error codes and their meanings.

Table 3
SABR Error Codes

Error Code	Meaning
A	Too many or too few ARG statements follow a call statement.
C	An illegal character appears on the line.
D	A device handler has returned a fatal condition.
L	/L or /G option was indicated, but the LOADER.SV file does not exist on the system device.
M	A symbol is multiply defined. Listing of programs with multiple definitions have unmarked errors.

(continued on next page)

Table 3 (Cont.)
SABR Error Codes

Error Code	Meaning
I	An illegal syntax has been used, (as one of the following): 1. a pseudo-op with improper arguments, 2. a quote mark with no argument, 3. a non-terminated text string, 4. an improper address, 5. an illegal combination of micro-instructions.
E	There is no END statement.
S	Either the symbol table has overflowed, common storage has been exhausted, more than 64 different user-defined symbols occurred in a core page, or more than 64 external symbols have been declared. Could also indicate a system error such as overflowed output file.
U	No symbol table is being produced, but there is at least one undefined symbol in the program.
UNDF	Undefined symbol, printed in the symbol table listing.

13.0 LINKING LOADER

The Linking Loader is the system program used to load and link your program and subprograms in memory. It can be called automatically to load or load and start a FORTRAN or SABR program, or called independently to load or load and start a relocatable binary file stored on a device. Capable of loading programs over itself, SABR has options which allow you to obtain storage map listings of core availability.

The Linking Loader can search program libraries for subroutines which are referenced by the program in core and load those subroutines needed. (A library is a collection of relocatable subroutines -- FORTRAN or SABR output -- with a directory at the beginning to facilitate searching.) Any library can be searched by using the /L option to the Loader, but the system library, LIB8.RL, is searched automatically just before the Loader completes the building of a core image of your program. If LIB8.RL is not on the system device, there is no automatic library search. (The system program LIBSET allows you to build your own subroutine library.)

The Linking Loader can load any number of user and library programs into any field of memory. Several programs are usually 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 200 upwards. The space reserved for common storage is subtracted from the available loading area in field 1. The program reserving the largest amount of common storage must be loaded first.

The Run-Time Linkage Routines necessary to execute SABR programs are automatically loaded into the required areas of every field by the Linking Loader as part of its initialization. You need to know which areas of core these routines occupy.

13.1 Calling and Using the Linking Loader

You can automatically call the Linking Loader following assembly of either a SABR program or a SABR-assembled FORTRAN program by use of the /L or /G option. For details on automatic calling of the Linking Loader, see the description of FORTRAN II.

When you want to call the Linking Loader to load or load and start a relocatable binary file, issue the command:

```
R LOADER
```

in response to the Keyboard Monitor dot. The Command Decoder replies by printing an asterisk in the left margin; you can then indicate input and output files and options. Zero to 1 output files and 1 to 9 input files are possible. Only one binary program per file is permitted. The assumed extension for input files is .RL. The output file, if indicated, is used to hold a map of the loaded program.

You can either specify all options and operations to be performed on one line or to have various operations performed individually. Where all options are being specified at one time, the line to the Command Decoder contains the complete instructions for the Linking Loader. If operations are to be done individually, you can type a command, enter it with the RETURN key, and that command will be executed, with another command expected when the first is completed. To indicate the last command, type an ALT MODE character, or end the last command with a /G option to start the program.

13.1.1 Linking Loader Options - The options to the Linking Loader are listed and explained in Table 4.

Table 4
Linking Loader Options

Option	Meaning
/I	A program doing device-independent input is to be loaded. (This feature costs the user 3 pages of core.)
/O	A program doing device-independent output is to be loaded. (This feature costs the user 3 pages of core.) If both /I and /O are indicated, 6 pages of core are used to handle deviice-independent I/O. /I and /O, if used, must be given before or on the first input line specifying files to be loaded. For example: *INPUT,FILES/O$ is acceptable, but *INPUT */O FILES is not legal and will generate an error message.
/H	A program doing device-independent I/O requires two-page device handlers at run-time. (This feature costs you one additional page if you are doing just input or output, and two additional pages if you are doing input and output. If /I, /O, and /H are indicated, 8 pages of core are used to handle device-independent I/O. /H, IF used, must be indicated on or before the first line containing /I or /O, and is meaningless without /I or /O also being specified.
/G	Start the program after processing the rest of the command string. Execution starts at the symbol MAIN unless otherwise indicated.
=n	Specifies the starting address of the program if other than the entry point MAIN; n is an octal number up to 5 digits long.

(continued on next page)

Table 4 (Cont.)
Linking Loader Options

Option	Meaning
/M	Output a map of the loaded programs onto the output file specified, followed by a count of the free pages in each field. If no output is specified, the map is put onto the teleprinter. The assumed extension for map output file is .MP. The map is printed after the rest of the command line is processed.
/U	Similar to /M, but only outputs undefined symbols.
/P	Similar to /M, but only outputs count of free pages in each field.
/n	Search through the available fields starting at field n for space large enough to hold each input file; n is an integer in the range 0 to 7, inclusive. Only one binary program can be in each input file. If n is not specified, the Loader starts looking at field 0.
/R	Restart loading process (forget all previously loaded programs). This command is equivalent to restarting the Linking Loader, but is much faster for DECtape systems since no tape motion is involved.
/L	Load the first input file as a library file (Loader expects a Library Directory as the first block of the file). All other input files on the line are ignored.

The Core Availability option (/P) causes the number of free pages of memory in every field of memory to be printed in a list on the teleprinter. For example, if you have a 16K configuration, a list like the following might be printed:

```
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 (/M), when selected, causes a list of all program entry points to be printed along with the actual address at which they have been loaded. Entry points of programs that have been called but that have not been loaded are also listed along with U flag for "undefined". Such flagged programs must be loaded before execution of your programs are possible. The core availability list is automatically appended to the storage map. A sample is shown below for an 8K machine:

```
MAIN        10200
READ        01050
WRITE       01066
IOH         03031
ERROR       00000 U
GENIO       00000 U
FDV         04722
CLEAR       05247
IFAD        05131
FMP         04632
ISTO        05074
STO         04447
FLOT        05210
FAD         04010
DIV         00000 U
IREM        00000 U
FSB         04000
FLOAT       05046
FIX         04513
IFIX        04561
CHS         05231
0011
0033
```

13.1.2 **Examples of I/O Command Strings** - Examples of possible input command strings follow.

```
*PROG,DTA2:SUB1,SUB2/G
```

This string loads DSK:PROG.RL, DTA2:SUB1.RL, DTA2:SUB2. RL, loads any necessary library routines requested, and starts the program at the entry point MAIN. The same process could have been done as follows:

```
Load DSK:PROG.RL;

     Get a list of undefined symbols on the teleprinter;

     *PROG

     */U
     .
     .     (Symbols go here)
     .
     .

     *DTA2:SUBR1,SUBR2
```

Load DTA2:SUBR1.RL,SUB2.RL;

*LPT:/M<$

Put loading map on the line printer, load the binary of any library routines requested by the program, and exit ($ is printed by the ALT MODE key);

.SAVE DTA2 FORTPG

Save the core image on DTA2 as FORTPG.SV;

Start the core image at its starting address (entry point MAIN in this case).

.START

13.2 Linking Loader Error Messages

The Linking Loader outputs error messages in the form

ERROR nnnn

where nnnn represents a 4-digit error code. Table 5 lists the error codes and their meanings.

Table 5
Linking Loader Error Messages

Error Code	Meaning
0000	/I or /0 specified too late.
0001	Symbol table overflow; more than 64 subprogram names.
0002	Program will not fit into core.
0003	Program with largest common storage area was not loaded first.
0004	Checksum error in input tape.
0005	Illegal relocation code.
0006	An output error has occurred.
0007	An input error has occurred (either a physical device error, or an attempt was made to read from a write-only device such as LPT:).
0010	No starting address has been specified and there is no entry point named MAIN.
0011	An error occurred while the Loader attempted to load a device handler.
0012	I/O error on system device.

14.0 LIBRARY SETUP (LIBSET)

LIBSET, the FORTRAN Library Setup program, creates a library of subroutines from the relocatable binary output of SABR. These library files can be quickly and effectively scanned by the Linking Loader, thus saving a great deal of the time involved in loading frequently used subroutines. (Refer to the section concerning the Linking Loader for information pertaining to relocatable library files, automatic loading of the LIB8.RL file, and the /L option.)

14.1 Calling and Using LIBSET

To call LIBSET from the system device, type

R LIBSET

in response to the Keyboard Monitor dot. The Command Decoder then prints an asterisk in the left margin of the teleprinter paper and waits to receive a line of input. The general form of input required to build a library file is:

```
*DEV:OUTPUT FILE<DEV:INPUT FILE(S)
*(additional input files) $
```

No more than nine input files are allowed on any one line, but several input lines can be entered. The last input line must end with your typing the ALT MODE key (which echoes as $). Only the first line can contain an output file. If no output file is specified, a file named LIB8.RL is created on the system device. The assumed extension for both input and output files is .RL.

NOTE

Files output from LIBSET are in a special relocatable library format and must not be copied with the /B option in PIP. Instead, they should be copied by PIP in image (/I) mode.

14.1.1 **LIBSET Options** - Only one option is allowed in the use of LIBSET, and this is described below:

Option	Meaning
/S	The /S option means that all input files on a line are to be regarded as containing more than one relocatable binary file. (This is analogous to the /S option in ABSLDR.)

NOTE

If /S is used on a line that contains no input files, input from PTR: is assumed.

14.1.2 Examples of LIBSET Usage

Example 1:

```
*DTA2:SUBS<DTA1:SUB1,SUB2,SUB3,PTR:
^*SYS:FUNC1,FUNC2.V5$
```

This example creates a relocatable library file on DTA2 named SUBS.RL. This library will contain six FORTRAN (or SABR) subroutines built by combining the relocatable binary file SUB1.RL, SUB2.RL, and SUB3.RL from DTA1 together with one relocatable binary paper tape (note the ^ printed by OS/8 before loading from PTR:) and the files FUNC1.RL and FUNC2.V5 from the system device.

Example 2:

```
*ASIN,ACOS
*/S$^
```

Since no output file was specified, this example creates a relocatable library file LIB8.RL on the system device. This produces a new FORTRAN library including the subroutines contained in the files ASIN and ACOS on device DSK, and several subroutines combined on a single paper tape loaded from the high-speed reader.

14.2 Subroutine Names

It is important to distinguish between the OS/8 file name of a relocatable binary program and its assigned Entry Point name. The file name has meaning only to the Command Decoder; the Entry Point name (or names) are the true subroutine names that are meaningful to the Loader.

Further details on the format of relocatable binary files and relocatable library files can be found in the OS/8 Software Support Manual (DEC-S8-OSSMA-A-D).

14.3 Sequence for Loading Subroutines

LIBSET can combine files in any sequence to form a relocatable library file. However, the subroutines in any single library are loaded by the Loader in the order in which they were originally specified to LIBSET. Therefore, it is important to make sure that subroutines are specified in order of size, with the largest subroutine being loaded first. If this is not done, cases can occur in which insufficient core is available in any single field to load a subroutine, whereas space would have been available if the subroutine had been loaded earlier.

14.4 LIBSET Error Messages

All errors are fatal. LIBSET recalls the Keyboard Monitor upon encountering an error condition and must be recalled in order to enter another command string. (See Table 6.)

Table 6
LIBSET Error Messages

Error Message	Meaning
BAD FORMAT OR CHECKSUM-- TRY AGAIN	Error in reading relocatable binary file.
ERROR WHILE WRITING OUTPUT FILE	Fatal output error occurred.
INPUT ERROR	Parity error on input.
LIBRARY DIRECTORY OVERFLOW	Too many subroutines were specified. Every subroutine name in the input file requires four words, and every relocatable binary file read requires two words. If the total number of words exceeds 250, the library must be split into two separate files.

15.0 LIBRARY PROGRAMS

During execution, the Library programs check for errors and type out error messages in the form:

XXXX ERROR AT LOC NNNN

where XXXX specifies the type of error, and NNNN is the 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 relevant to subprograms called by the statement at the relevant location. (See Table 7.)

Table 7
Library Error Messages

Error Message	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; attempt to fix too large a number.
SQRT	Attempt to take root of a negative number

OS/8 includes, in addition, the error message:

USER ERROR 1 AT 00537

which means that you tried to reference an entry point of a program that was not loaded.

To pinpoint the location of a Library execution error, proceed as follows.

1. 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.

2. 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.

3. From the assembly symbol table, determine the relative address of the external symbol found in step 1 and add that relative address to the result of step 2.

4. The sum of step 3 is the relative address of the error, which can then be compared with the relative addresses of the numbered statements in the program.

16.0 DEMONSTRATION PROGRAM USING LIBRARY ROUTINES

The following demonstration program is a SABR program showing the use of the library routines. The program was written to add two integer numbers, convert the result into floating-point, and type the result in both integer and floating-point format. The source program was written using the Symbolic Editor, assembled with SABR, and loaded with the Linking Loader, under the OS/8 Operating System.

```
A       0257
B       0260
C       0261
D       0262
FLOAT   0000EXT
FORMT   0240
IOH     0000EXT
N       0256
OPEN    0000EXT
START   0200EXT
STO     0000EXT
WRITE   0000EXT
```

```
                        ENTRY START

0200    4033     START, CALL 0,OPEN      /INITIALIZE
0201    0002 06
                                         /I/O DEVICES
0202    1257           TAD A             /COMPUTE C=A+B
0203    1260           TAD B
0204    3261           DCA C
0205    4033           CALL 1,FLOAT      /CONVERT TO
0206    0103 06
                                         /FLOATING POINT
0207    6201 05        ARG C
0210    0261 01
0211    4033           CALL 1,STO
```

```
0212   0104 06
0213   6201 05          ARG D
0214   0262 01
0215   4033             CALL 2,WRITE     /INITIALIZE
0216   0205 06
                                         /I/O HANDLER
0217   6201 05          ARG N            /DEVICE NUMBER
0220   0256 01
                                         /1=TELETYPE
0221   6201 05          ARG FORMT        /FORMAT SPECI-
0222   0240 01
                                         /FICATION
0223   4033             CALL 1,IOH       /TYPE INTEGER
0224   0106 06
                                         /NUMBER
0225   6201 05          ARG C
0226   0261 01
0227   4033             CALL 1,IOH       /TYPE FLOATING
0230   0106 06
                                         /POINT NUMBER
0231   6201 05          ARG D
0232   0262 01
0233   4033             CALL 1,IOH       /COMPLETE THE I/O
0234   0106 06
0235   6211             ARG 0
0236   0000
0237   7402             HLT

0240   5047      FORMT, TEXT "('THE ANSWERS ARE',I5,F7.2)"
0241   2410
0242   0540
0243   0116
0244   2327
0245   0522
0246   2340
0247   0122
0250   0547
0251   5411
0252   6554
0253   0667
0254   5662
0255   5100
0256   0001      N,     1
0257   0002      A,     2
0260   0002      B,     2
0261   0000      C,     0
0262   0000      D,     BLOCK 3
0263   0000
0264   0000

                 END
```

The binary tape produced by the assembly was then run using OS/8 with the following results:

```
THE ANSWERS ARE    4   4.00
```

APPENDIX

SABR INSTRUCTION CODES AND PSEUDO-OPERATORS

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

INSTRUCTION CODES

Mnemonic	Code	Operation	Time (mn sec.)*
Memory Reference Instructions			
AND	0000	Logical AND	2.6
TAD	1000	Two's complement add	2.6
ISZ	2000	Increment and skip if zero	2.6
INC	2000	Nonskip ISZ	2.6
DCA	3000	Deposit and clear AC	2.6
JMS	4000	Jump to subroutine	2.6
JMP	5000	Jump	1.2

Mnemonic	Code	Operation	Sequence
Group 1 Operate Microinstructions (1 cycle**)			
NOP	7000	No operation	--
IAC	7001	Increment AC	3
RAL	7004	Rotate AC and link left one	4
RTL	7006	Rotate AC and link left two	4
RAR	7010	Rotate AC and link right one	4
RTR	7012	Rotate AC and link right two	4
CML	7020	Complemented link	2
CMA	7040	Complement AC	2
CLL	7100	Clear link	1
CLA	7200	Clear AC	1

* Times are representative of the PDP-8/E.

** 1 cycle is equal to 1.2 microseconds.

Group 2 Operate Microinstructions (1 cycle)

HLT	7402	Halts the computer	3
OSR	7404	Inclusive OR SR with AC	3
SKP	7410	Skip unconditionally	1
SNL	7420	Skip on nonzero link	1
SZL	7430	Skip on zero link	1
SZA	7440	Skip on zero AC	1
SNA	7450	Skip on nonzero AC	1
SMA	7500	Skip on minus AC	1
SPA	7510	Skip on positive AC (zero is positive	1

Combined Operate Microinstructions

CIA	7041	Complement and increment AC	2,3
STL	7120	Sent link to 1	1,2
STA	7240	Set AC to -1	2

Internal IOT Microinstructions

ION	6001	Turn interrupt processor on
IOF	6002	Disable interrupt processor

Keyboard/Reader (1 cycle)

KSF	6031	Skip on keyboard/reader flag
KRB	6036	Clear AC, read keyboard buffer (dynamic), clear keyboard flags

Teleprinter/Punch (1 cycle)

TSF	6041	Skip on teleprinter/punch flag
TLS	6046	Load teleprinter/punch, print, and clear teleprinter/punch flag

High Speed Reader -- Type PR8/E (1 cycle)

RSF	6011	Skip on reader flag
RRB	6012	Read reader buffer and clear reader flag
RFC	6014	Clear flag and buffer and fetch character

High Speed Punch -- Type PP8/E (1 cycle)

PSF	6021	Skip on punch flag
PLS	6026	Clear flag and buffer, load buffer and punch character

PSEUDO-OPERATORS

The following is a list of the SABR assembler pseudo-operators.

ABSYM
ACH
ACM
ACL
ARG
BLOCK
CALL
COMMN
CPAGE
DECIM
DUMMY
EAP
END
ENTRY
FORTR
I
IF
LAP
OCTAL
OPDEF
PAGE
PAUSE
REORG
RETRN
SKPDF
TEXT

INDEX

OS/8
Language Reference Manual
AA-H609A-TA

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

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

Did you find errors in this manual? If so, specify the error and the page number.

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

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)_____

Name_____ Date_____

Organization_____

Street_____

City_____ State_____ Zip Code or Country_____

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

RT/C SOFTWARE PUBLICATIONS ML 5-5/E45
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET
MAYNARD, MASSACHUSETTS 01754

Do Not Tear - Fold Here

Cut Along Dotted Line