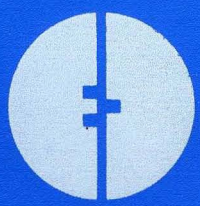
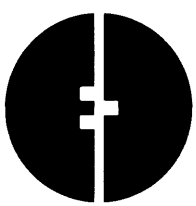
NEVADA FORTRAN



ELLIS COMPUTING

SOFTWARE TECHNOLOGY

NEVADA FORTRAN



ELLIS COMPUTING

SOFTWARE TECHNOLOGY

NEVADA FORTRAN (TM)

PROGRAMMER'S REFERENCE MANUAL

Published by

ELLIS Computing
3917 Noriega Street
San Francisco, CA, 94122
(415) 753-0186

DISCLAIMER

All Ellis Computing computer programs are distributed on an "AS IS" basis without warranty.

ELLIS COMPUTING makes no warranties, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. In no event will ELLIS COMPUTING be liable for consequential damages even if ELLIS COMPUTING has been advised of the possibility of such damages.

TABLE OF CONTENTS

ABSTRACT

This is an 8080/8085/Z80 version of FORTRAN IV. It is a powerful subset implementation of this widely used language. The compiler works from disk (also using the assembler) to produce 8080/8085/Z80 machine code that executes at maximum CPU speed.

A source program is entered as FORTRAN IV program statements. These statements must follow the conventions outlined in this document or errors may result. The compiler acts upon the source statements to produce assembly code. At this stage any mistakes are flagged with error messages. If an error should occur, the source may be corrected at this time and recompiled. After the program has been compiled without any errors, the final step (normally transparent to the user) is to assemble the intermediate code into 8080 object code. The object module is then ready for execution under CP/M (or compatible operating system).

This manual is intended as a guide to using this version of FORTRAN and is not intended to be a complete instruction manual on the use of FORTRAN. There exists many books that explain the syntax and semantics of the FORTRAN language. This manual explains the subset that is implemented in Nevada FORTRAN.

1.0.0 FORTRAN LANGUAGE

1.1.0 FORTRAN Character Set

The FORTRAN character set is composed of the following characters:

The letters:

A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z

The numbers:

0,1,2,3,4,5,6,7,8,9

The special characters:

	blank
=	equal sign (for replacement operations)
+	plus sign
-	minus sign
*	asterisk
/	slash
(	left parenthesis
)	right parenthesis
,	comma
.	decimal point
$	dollar sign
#	number sign
&	ampersand
\	backslash

The following is a list of the meanings of the special characters used in <u>this</u> version of FORTRAN:

$ Preceding a constant with a dollar sign indicates that it is a hexadecimal constant.

Preceding a constant with a number sign indicates that it is a hexadecimal constant that is to be stored internally in binary format.

& The & has two functions:
1) if used in a FORMAT statement contains an ampersand, the character following the ampersand is interpreted as a control character (unless it is also an &).
2) used to indicate a statement label is being passed to a SUBROUTINE for use in a multiple RETURN statement

\ A constant enclosed in backslashes in a character string is assumed to be the hexadecimal code for an ASCII character.

1.2.0 Program Structure

A FORTRAN program is comprised of statements. Every statement must be of the following format.

1) The first 5 characters of the statement may contain a statement label if the statement is to be branched to.

2) The sixth character is used to indicate a continuation of the previous statement. Continuation is indicated by placing any character except a BLANK or ZERO in column 6 of the continuation statement.

3) Column 7 to the end of the line is used for the body of the statement. This is any one of the following statements which will be described later. All statements are terminated by a CR (Carriage Return) or semi-colon in the case of multiple statements per line. A statement may be of any length, but only the first 72 characters are retained. Statements will be processed until the carriage return is encountered. The character positions between the carriage return and character position 72 will **NOT** be padded with BLANKS as some FORTRAN systems will do, unless the '1' or '2' option is specified. This means that if a character string is started on a line, and must be continued, the continuation logically starts immediately after the last character of the previous line.

4) Column 73 through 80 are used for identification purposes and are ignored by this compiler.

5) A comment line is indicated by place a **C** in column 1. A comment line has no effect on the program and is ignored. It is only for documentation purposes.

Example

```
|-- Column 1
|
V

      WRITE (1,2)
    2 FORMAT ('THIS IS AN
     * EXAMPLE CHARACTER STRING')
```

will output: THIS IS AN EXAMPLE CHARACTER STRING

Upper and lower case letters can be intermixed in a FORTRAN statement. Lower case letters are retained ONLY when they appear between QUOTES or in the **H** format specification in a FORMAT statement. Otherwise they will be converted to upper case internally. Thus the variables **QUANTITY** and **quantity** and **QuAnTiTy** represent the same variable.

There are four types of statements in FORTRAN:

1) Declaration
2) Assignment
3) Control
4) Input/Output.

These statement types are described in the following sections of this manual.

1.3.0 Statements

A statement may contain a statement label. A statement label is placed in columns 1 through 5 of the statement.

All labels on statements must be integers ranging between 1 and 99999. Leading zeroes will be totally ignored.

Statement numbers are not required to be in any sequence, and they will not be put in order.

In any program, a statement number can be used only once as a label.

A statement may contain no more than 530 characters excluding blanks, unless the **B=** option is specified.

During compilation, blanks are ignored, except between single quotes and in **H** format specification.

Comments are indicated by placing a C in column 1; the remaining part of the statement may be in any format and is totally ignored by the compiler.

Multi-statements

Statements may be compacted more than one logical statement per line. Statements are separated from each other with a semicolon and a colon separates the label if any. For example;

```
    A=1
  3 CONTINUE
    A=A+1
    TYPE A
    GO TO 3
    END
```

could be written as:

```
    A=1;3:CONTINUE;A=A+1;TYPE A:GOTO 3;END
```

1.4.0 Program Preparation

A FORTRAN source program is prepared using one of the available CP/M text editors. The FORTRAN file must be in the following format:

```
Position 1...5....0....5....0....5....0....5....0....5
         OPTIONS
                :
                :
                :
           FORTRAN program
                :
                :
               END
        $OPTIONS
               SUBROUTINE X
                :
            FORTRAN routine
                :
               END
```

All FORTRAN routines are **required** to be compiled at one time.

A FORTRAN program can contain COPY statements. The COPY statement contains the word **COPY** in columns 7-10 followed by a blank followed by the FILENAME to be inserted at that point. COPY files may contain complete programs or just sections. Copied files may not themselves contain COPY statements.

Example

```
      DIMENSION A(1)
      COPY ALLDEFS
      READ (1,10) I
```

1.5.0 Options Statement

$OPTIONS or OPTIONS (The $ is optional)

This is an optional statement of each program and/or subprogram which is to be compiled. If present, the OPTIONS statement must appear as the first statement in main program and prior to the SUBROUTINE or FUNCTION statement in each subprogram. The options statement allows the user to specify various parameters used by the compiler during compilation. Options available are as follows:

S=n

n -> Is a decimal number indicating the number of allowable symbols. The default is 50. Each entry requires 8 bytes. (n may be greater than 255.)

L=n

n -> Is a decimal number indicating the number of allowable labels. The default is 50. Each entry requires 6 bytes. (n may be greater than 255.)

T=n

n -> Is a decimal number indicating the maximum number of temporary variables that are available during EXPRESSION evaluation. Default table size is 15; each variable requires 1 byte.

D=n

n -> Is a decimal number which indicates the maximum allowable nesting of DO loops. Default is 5, each entry requires 4 bytes.

A=n

n -> Is a decimal number which indicates the maximum number of arrays. Default is a maximum of 15; each entry requires 4 bytes.

O=n

n -> Is a decimal number which indicates the maximum number of operators ever pushed on the internal stack while doing a prefix translation of input expression. Note functions and array subscripting require a double entry. Default is 40; each entry is 2 bytes long.

P=n

n -> Is a decimal number which indicates the maximum number of variables and/or constants ever pushed on the internal stack in evaluation. Default is 40; each entry is 2 bytes long.

I=n

n -> Is a decimal number specifying the depth that IF-THEN-ELSE's may be nested. The default nesting is 5.

E

Instructs the compiler to list as comments a reference table equating user symbols, constants, and labels to internally generated ones.

G

Instructs the compiler to list all compile errors as error numbers, instead of explicit error statements.

X

Instructs the compiler to generate code which will give explicit runtime errors. In this mode each statement has 5 bytes overhead.

N

Check for FORTRAN errors only. Do not output an assembly code file.

B

Output source statement to assembly file.

Q

This options **must** be used whenever the program expects to trap runtime errors. It causes code to be generated for handling user trapping of runtime errors.

n is less than or equal to 255 unless otherwise stated.

Example

```
$OPTIONS X,G,S=200,L=100
```

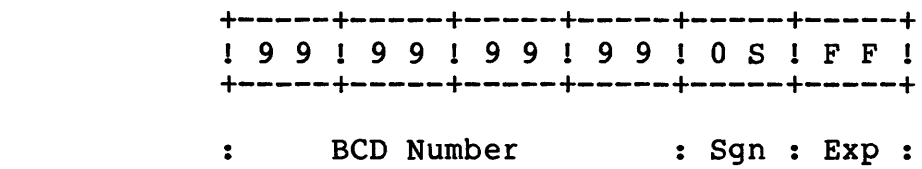
Options used will be:
EXPLICIT runtime errors will be generated
EXPLICIT compile errors are not generated
the SYMBOL table has room for 200 symbols, and
the LABEL table has room for 100 statement labels.

2.0.0 NUMBER SYSTEM

2.1.0 Internal Format of Numbers

Numbers are stored internally as a 6 byte BCD number containing 8 digits, a one byte exponent, and a sign byte. This allows for the number to range from .1E-127 to number; 0 indicating a positive number and 1 indicating a negative number. The exponent is stored in excess 128. A one for the sign of the BCD number indicates a negative number. The number ZERO is stored as an exponent of zero; the rest of the number is ignored.

All numbers in FORTRAN are stored in the following format:



2.2.0 Number Ranges

Integer variables and constants can have any value from -99999999 to +99999999. Real variables and constants can take any value between -.99999999E-127 and .99999999E+126. Integer variables and constants are stored internally in the same format.

2.3.0 Constants

A constant is a quantity that has a fixed value. A numerical constant is an integer or real number; a string constant is a sequence of characters enclosed in single quotes. A logical constant has a value of .TRUE. or .FALSE.

2.3.1 Numerical Constants

Numerical constants can be either integer or real as follows:

Integer	1, 3099, -70
Real	1.34, -5.98, 1.4E10

A hexadecimal constant can be specified by preceding the number with a dollar sign. A hexadecimal constant is converted internally into an integer and stored that way. The maximum value for a hexadecimal constant is $FFFF.

Example

```
$8050
I=$1000
z=-$CC00
```

Another way to specify a hexadecimal constant is to preceded the constant with a # sign. This way of representing a hexadecimal number differs in that the number is NOT converted to integer format and is stored in binary in the first two bytes of the constant. The number is stored high byte followed by low byte.

Example

```
#0D00
i=#127F

$805F is stored internally as: 32 86 30 00 00 85
#805F is stored internally as: 5F 80 00 00 00 00
```

2.3.2 String Constants

A string constant is specified by enclosing a sequence of characters in single quotes. A single quote within a character string must be represented by **TWO** quotes in a row. By specifying a hexadecimal number with backslashes, any character (even unprintable ones) can be generated.

Example

```
'This is a string constant'
'This string constant''contains a single quote'
'Good\21\'  is equivalent to 'Good!'
'\7F\' is equivalent to a rubout
```

Warning: Never include \0\ as part of a string constant as that character is used internally to indicate the end of a string.

NOTE The character used to delimit a hexadecimal number (default is \) can be changed using the CONFIG program.

2.3.3 Logical constants

The two logical constants are .TRUE. and .FALSE.. Numerically, a value of .TRUE. has the value of 1 and considered as .TRUE. Logical operations **always** return a value of 0 or 1. These logical constants can be assigned to any variable, but is usually used as part of a logical expression.

Example

```
I=.TRUE.
I=(J .and. .TRUE)
```

2.4.0 Variable Names

A variable is a symbolic name given to a quantity which may change depending upon the operation of a program. A variable consists of from 1 to 6 alphanumeric characters, the first of which **must** be a letter.

An INTEGER variable is a variable that starts with I, J, K, L, M or N by default or explicitly typed INTEGER through the use of an INTEGER or IMPLICIT statement.

A REAL variable is a variable that starts with other than I, J, K, L, M or N by default or explicitly typed REAL through the use of a REAL or IMPLICIT statement.

A LOGICAL variable must be explicitly typed LOGICAL with a LOGICAL or IMPLICIT statement.

There are three types of variables supported: **INTEGER, REAL,** and **LOGICAL.**

Example

```
I10, ALPHA, BETA, I, MAXIM, MINII
IX=34
ALPHA=56.34
ZLOG=.TRUE.
```

2.5.0 Type Specification

There are three type specification statements that can be used to override the default types of variables. Remember that variables that begin with the letters I,J,K,L,M,N (unless changed by an IMPLICIT statement) will be of type INTEGER. All others will be of type REAL. The type specification statement overrides the default type of a variable.

Note an array can also be specified in a type statement.

Example

```
INTEGER A,ZOT,ZAP(10)
REAL INT
LOGICAL LOG1,LOG2
```

2.5.1 INTEGER

The general format of the INTEGER statement is:

INTEGER v1,v2

The INTEGER statement is used to explicitly override the default type of the variable. Should a variable occur in the declaration string, the type is automatically set to integer. This works for both subscripted and nonsubscripted variables. A variable can appear only ONCE in a type specification statement.

Example

```
INTEGER MODE,K453,NUMBER(40),MAXNUM
INTEGER ZAPIT
```

2.5.2 REAL

The general format of the REAL statement is:

REAL v1,v2

The REAL statement is used to explicitly override the default type of the variable. Should a variable occur in the declaration string, the type is automatically set to real. This works for both subscripted and nonsubscripted variables. A variable can appear only ONCE in a type specification statement.

Example

```
REAL ALPHA,BETA(56),INIT,FIRST,ZAPIT,HI
```

2.5.3 LOGICAL

The general format of the LOGICAL statement is:

LOGICAL v1,v2

The Logical statement is used to override the default specification and type a variable as Logical. A logical variable's value is interpreted as:

.TRUE. if the variable has a non-zero value.
.FALSE. if the variable has a zero value.

Example

```
LOGICAL FTIME,LTIME
```

2.6.0 Data Statement

The DATA statement is used to initialize variables or arrays to a numeric value or character string. The general format is:

DATA list/n1,n2..../,list1/n1,n2/

where **list** is the list of variables (or array elements) to be initialized and n1, n2.. are numbers or strings (constants) that the corresponding item of **list** will be initialized to. An exception to this is the array name. If only the name of the array (no subscripts) appears in **list**, the whole array will be initialized. It is expected that enough constants will be listed to completely fill the array. Dummy subroutine arguments may not appear in **list**. A restriction exists on DATA statements in that all the DATA statements are stored in memory during compilation of a particular routine and actually compiled after the END statement of the that routine. As such, you may receive a fatal error if you use more memory than available for storing of the DATA statements and error messages will be printed after the END statement of that routine.

Example

```
DIMENSION B(3),C(3)
DATA A/1/,B/1,2,3/,C/3*0/
DATA LIST/'THIS IS A CHARACTER STRING'/
```

The above statement will assign the value 1 to A and the values 1 to B(1), 2 to B(2) and 3 to B(3). The asterisk is used to indicate a repeat count; thus the array C will be set to zeroes. An error will result if a variable in a DATA statement is not used elsewhere in a program. DATA statements are processed when the END statement is encountered; thus errors in a DATA statement will occur after the END statement. These errors will include the four digit FORTRAN assigned statement number and the variable in the DATA statement being processed when the error occurred.

2.7.0 Common Blocks

The COMMON block declaration sets aside memory (variable space) to be shared between routines (SUBROUTINES, FUNCTIONS). Common blocks are associated with a name which is used by each declaring routine to point to a specific COMMON block.

The general form of a COMMON statement is:

COMMON /name1/ list1 /name2/ list2

where **name1** and **name2** are the COMMON block names associated with the corresponding **list1** and **list2.**

Example

```
DIMENSION X(100)
COMMON /ZZZ/ FIRST,LAST,X
CALL ADDEM
    .
    .
END

SUBROUTINE ADDEM
REAL NUMBER
COMMON /ZZZ/ F,L,NUMBER(100)
    .
    .
END
```

An array declaration may be included in a COMMON statement as shown in the subroutine. The use of common blocks allows data to be passed to and from a subprogram, but without passing it as arguments (in a heavily called routine, this method can save execution time). If an array is to be included in a common declaration, it must either be declared previously or declared in the COMMON statement.

If the name is omitted or the name is null (i.e. //) then it is called blank COMMON.

Example

```
COMMON A,B,C,D
COMMON // A,B,C,D    are equivalent statements
```

Blank COMMON differs from named COMMON in the following ways:

1) variables in blank common are allocated their actual memory addresses at runtime and therefore cannot be initialized with a DATA statement.

2) blank common is allocated at runtime directly below the TPA in CP/M or at a user specified address (see M= parameter when program is compiled). If the size of blank common blocks is the same, then blank common can be used to pass data between routines that CHAIN as the blank common variables will occupy the same place in memory.

NOTE: The name of a named COMMON block can not be the same as a SUBROUTINE or FUNCTION name.

2.8.0 Implicit Statement

The IMPLICIT statement is used to change the default integer, real and logical typing.

The general format of the IMPLICIT statement is:

IMPLICIT type (range), type(range)

where:

Type is one of INTEGER, REAL or LOGICAL. **Range** is either a single letter or a range of letters in **alphabetical** order. A range is denoted by the first and last letter of the range separated by a hyphen or a sequence of single letters separated by commas.

Example

```
IMPLICIT INTEGER (Z),REAL (A,B,C,D,E,G),INTEGER (M-S)
IMPLICIT REAL (I,J)
IMPLICIT REAL (A-Z)
```

An IMPLICIT statement specifies the type of all variables, arrays and functions that begin with any letter that appears in the specification. Type specification by an IMPLICIT statement may be overridden for any particular variable, array or function name by the appearance of that name in a type statement.

The IMPLICIT statement **must** appear before all other statements in a particular routine: that is immediately after the SUBROUTINE or FUNCTION statement or before the first statement of the main program.

If the name is omitted or the name is null (i.e. //) then it is called blank COMMON.

Example

```
COMMON A,B,C,D
COMMON // A,B,C,D   are equivalent statements
```

Blank COMMON differs from named COMMON in the following ways:

1) variables in blank common are allocated their actual memory addresses at runtime and therefore cannot be initialized with a DATA statement.

2) blank common is allocated at runtime directly below the TPA in CP/M or at a user specified address (see M= parameter when program is compiled). If the size of blank common blocks is the same, then blank common can be used to pass data between routines that CHAIN as the blank common variables will occupy the same place in memory.

NOTE: The name of a named COMMON block can not be the same as a SUBROUTINE or FUNCTION name.

2.8.0 Implicit Statement

The IMPLICIT statement is used to change the default integer, real and logical typing.

The general format of the IMPLICIT statement is:

IMPLICIT type (range), type(range)

where:

Type is one of INTEGER, REAL or LOGICAL. **Range** is either a single letter or a range of letters in **alphabetical** order. A range is denoted by the first and last letter of the range separated by a hyphen or a sequence of single letters separated by commas.

Example

```
IMPLICIT INTEGER (Z),REAL (A,B,C,D,E,G),INTEGER (M-S)
IMPLICIT REAL (I,J)
IMPLICIT REAL (A-Z)
```

An IMPLICIT statement specifies the type of all variables, arrays and functions that begin with any letter that appears in the specification. Type specification by an IMPLICIT statement may be overridden for any particular variable, array or function name by the appearance of that name in a type statement.

The IMPLICIT statement **must** appear before all other statements in a particular routine: that is immediately after the SUBROUTINE or FUNCTION statement or before the first statement of the main program.

3.0.0 Expressions

An expression is a combination of variable, functions and constants, joined together with one or more operators.

Arithmetic Operators

** or ^	Exponentiation
*	Multiplication
/	Division
+	Addition
-	Subtraction

Comparison Operators

.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.LT.	Less than
.GE.	Greater than or equal
.LE.	Less than or equal to

Logical Operators

.NOT.	Logical Negation
.AND.	Logical and
.OR.	Logical or
.XOR.	Logical exclusive or

The .NOT. and unary minus (-) operators preceded an operand. All other operators join two operands.

3.1.0 Hierarchy of Operators

The following is the table of operator hierarchy and the correct FORTRAN symbolic representation to be used in coding:

Highest	System and User Functions
	** OR ^ (up arrow)
	* and /
	+ and - (including unary -)
	.LT., .LE. ,.NE. ,.EQ. ,.GE. ,.GT.
	.NOT.
	.AND.
	.OR. and .XOR.
	Replacement (=)
Lowest	

3.2.0 Expression Evaluation

FORTRAN expressions are evaluated as follows:

1. Parenthesised expressions are always evaluated first, with the inner most set being evaluated first.

2. Within parentheses (or whenever there are none) the order of expression evaluation is:

   a. FUNCTION references
   b. Exponentiation
   c. Multiplication and division
   d. Addition and subtraction

Example

A+1+Z*5 will be evaluated as:

((A+(1+(Z*5))

VAL*Z+(T+4)/6*X**Y will be evaluated as:

((VAL*Z)+(((T+4)/6)*(X**Y)))

3.3.0 Integer Operations

A fundamental difference between integer and real arithmetic operation, is the manner in which rounding occurs. If you were to divide 3.0 by 2.0 using floating point arithmetic, the answer would be 1.5. However, if the same operation were to be performed using integer arithmetic, 3/2 would equal 1. Note in using integer arithmetic, the fractional part of the number is truncated. Another example would be in the multiplication of two real numbers. 2.9 times 4.8 would equal 13.92. However in integer mode, the result would be 13. Also, no more than 8 digits of accuracy are maintained. Should more than 8 digits be generated by an integer operation, a runtime error will result.

Example

```
      6/3=2
but   7/3=2 (NOTE: no fraction is retained) and  7/9=0
      99999999+5=?  integer overflow
```

3.4.0 Real Operations

Unlike integers, Real operations and their results have a precision of eight significant digits plus an exponent (base 10) between -127 and +127.

Example

```
12/6.0=2.0
15.0/2=7.5
 1./2.=0.5
```

3.5.0 Logical Operations

Logical operations are unlike integer and real operations in that they always return a value of zero (0) or one (1). All the logical operations will return a one for a TRUE condition, however any NON-ZERO value will be interpreted as TRUE. If the logical operation is a logically true statement, the result is a one, if the statement is false, a zero is returned.

Example

```
A = 1 .GT. 2   (false)  A would evaluate to 0
A = 1 .EQ. 1   (true)   A would evaluate to 1
A = 1 .LT. 2   (true)   A would evaluate to 1
```

The relational operator abbreviations in the previous table represent the following operations:

.LT.	Less Than
.LE.	Less Than or Equal
.NE.	Not Equal
.EQ.	Equal
.GE.	Greater Than or Equal
.GT.	Greater Than
.AND.	True only if both operands are true.
.OR.	True if either operand is true.
.XOR.	True if operands are different.

Example

```
IF (A .EQ. B) GO TO 500
IF (A .EQ. B.OR.K .EQ. D)STOP
```

Logical variables can also be used in assignment statements:

```
A=A .AND. B
I=(A .OR. B).XOR.((T .EQ. 35.4).OR.(T .EQ. 39))
```

The following logical operators are also available, as listed in the following truth charts.

.AND.

A	B	R
0	0	0
0	1	0
1	0	0
1	1	1

.OR.

A	B	R
0	0	0
0	1	1
1	0	1
1	1	1

.XOR.

A	B	R
0	0	0
0	1	1
1	0	1
1	1	0

.NOT.

A	R
0	1
1	0

3.6.0 Mixed Expressions

The standard FORTRAN rules for mixed mode expressions are:

integer <*op> **integer** gives an **integer** result

real <*op> **integer** gives a **real** result (with the **integer** being converted to **real** before the operation is performed).

real <*op> **real** gives a **real** result

integer <*op> **real** gives a **real** result, with the **integer** being converted to REAL before the operation is performed.

integer = real will cause truncation of any fractional part of **real** and an error if the truncated result is outside the range of integers.

real = integer will cause **integer** to be converted to **real.**

In general, in a mixed expression, **integers** are converted to **real** before the operation take place, giving a **real** result (unless both operands are **integer**).

4.0.0 Control Statements

There are several different statements that control the execution flow of a FORTRAN program:

These are:

1. GO TO statements
   a. Unconditional GO TO
   b. Computed GO TO
   c. Assigned GOTO

2. If statements
   a. Arithmetic IF
   b. Logical IF
   c. IF THEN ELSE

3. DO

4. CONTINUE

5. PAUSE

6. STOP

7. CALL

8. RETURN
   a. explicit RETURN
   b. multiple RETURN

4.1.0 Unconditional GO TO Statement

The general format of the unconditional GO TO is:

GO TO n

where **n** is a label on an executable statement.

The unconditional GO TO Statement performs an absolute transfer of control to the statement number specified as the object of the branch. If the statement number does not exist, an undefined label error will occur; this error is detected during compilation.

Note: Labels on format statements in most FORTRAN systems may not receive transfer of control. This is not true in this implementation of FORTRAN. Format statements act the same as a CONTINUE statement which will be discussed later.

Example

```
      GO TO 10
      GO TO 400

10    CONTINUE
400   FORMAT (1X)
```

4.0.0 Control Statements

There are several different statements that control the execution flow of a FORTRAN program:

These are:

1. GO TO statements
   a. Unconditional GO TO
   b. Computed GO TO
   c. Assigned GOTO

2. If statements
   a. Arithmetic IF
   b. Logical IF
   c. IF THEN ELSE

3. DO

4. CONTINUE

5. PAUSE

6. STOP

7. CALL

8. RETURN
   a. explicit RETURN
   b. multiple RETURN

4.1.0 Unconditional GO TO Statement

The general format of the unconditional GO TO is:

GO TO n

where **n** is a label on an executable statement.

The unconditional GO TO Statement performs an absolute transfer of control to the statement number specified as the object of the branch. If the statement number does not exist, an undefined label error will occur; this error is detected during compilation.

Note: Labels on format statements in most FORTRAN systems may not receive transfer of control. This is not true in this implementation of FORTRAN. Format statements act the same as a CONTINUE statement which will be discussed later.

Example

```
      GO TO 10
      GO TO 400

10    CONTINUE
400   FORMAT (1X)
```

4.2.0 Computed GO TO Statement

The general format of the COMPUTED GO TO is:

GO TO (nl,n2,...nm),i

The computed GO TO statement works in a manner similar to the GO TO statement. However, one of the distinct advantages is that under program control, you may direct which is the next instruction to be executed, based on the value of **i**. The computed GO TO works as follows:

Computed GO TO Statement	Present Value of Variable	Next Executed Statement
GO TO (1,5,98,167,4),K2	K2=5	4
GO TO (44,28),J	J=1	44
GO TO (51,6,7,1,46),M	M=4	1
GO TO (1,1,1,1,2,2),LOOT	LOOT=3	1

If the value of **i** exceeds the number of statement labels in the computed GOTO, then a runtime error is produced. If the value of **i** is less than 1, a runtime error is produced.

4.3.0 Assigned GO TO

The general format of the ASSIGNED GO TO is:

GO TO v,(nl,n2,...)

where **v** is the variable used in an ASSIGN statement and **nl, n2** are statement labels.

Example

```
GO TO LABL,(100,400,500)
GO TO K,(1,2,3,4,5)
```

4.4.0 ASSIGN

The general format of the ASSIGN statement is:

ASSIGN n TO V

where **n** is the statement label to be ASSIGNed to **v.** The ASSIGN statement assigns a statement label to be used in conjunction with the ASSIGNED GO TO statement.

Example

```
ASSIGN 20 TO LABEL
.
.
IF (KNT .GT. 10)ASSIGN 10 TO LABEL
.
.
GO TO LABEL,(10,20)
```

4.5.0 Arithmetic IF Statement

The Arithmetic IF allows the programmer to evaluate an expression which may be any combination of integer, real, or logical operators, and based on its relationship to zero perform a transfer of control operation.

The general form of the arithmetic IF is:

IF (e) n1,n2,n3

where a is an arithmetic expression which when evaluated is used to determine the next statement to be executed.

If e is:	next statement
<0	n1
=0	n2
>0	n3

Example

```
IF (A) 1,2,3
IF (BETA*SIN(BETA/DEGREE))100,150,432
IF (A-1)1,1,99
```

4.6.0 Logical IF Statement

The general form of the Logical IF is:

IF (e) s

The logical IF statement operates as follows:

1. The expression **e** is evaluated, and a logical result is derived, zero or one.

2. Depending on the value which is derived, one of the following two conditions occurs:

If **e** is evaluated as .TRUE. then the statement **s** is executed, and once the IF has completed, transfer is then passed to the next consecutive statement.

If **e** is evaluated as .FALSE. the statement **s** is not executed and control is then passed to the next sequential executable statement.

Example

```
IF (DEGREE .EQ. 100)WRITE (1,*) RADIAN
IF ((A .EQ. 12).OR.(LOOP .LE. 500))RETURN
IF (SIN(30)/WHERE-.00005 .LT. .00004)STOP
IF (A .NE. B)GO TO 500
IF (A .EQ. 1)GO TO (1,2,3),J
IF (VALUE .EQ. 6)IF (J)99,33,67
IF (1 .NE. 0)CONTINUE
```

4.7.0 IF-THEN-ELSE

The general format of this statement is

```
IF (e) THEN
        statement 1
        statement 2
        ...
        ELSE
        statement 3
        statement 4
        ...
   ENDIF
```

The IF-THEN-ELSE is an extension of the logical IF with 2 additions:

1) there can be more than 1 statement to execute if the IF is true

2) there is the provision of specifying one or more statements to be executed if the IF is false.

The ENDIF is required to indicate the end of the complete IF-THEN-ELSE statement.

To indicate an IF-THEN-ELSE the s part of the logical IF is replaced with the THEN statement. All statements between the THEN and the matching ELSE or ENDIF will be executed if the specified condition is true. All statements between the ELSE and ENDIF will be executed if the specified condition is false. The ELSE is optional and if the condition is false, all statements between the THEN and ENDIF will be skipped.

If no ELSE condition is to be specified, then the THEN can be terminated with an ENDIF. For example:

```
If (e) THEN
        statement 1
        statement 2
        ENDIF
```

Note: THEN, ELSE and **ENDIF** are individual statements terminated by either a carriage return or semicolon.

Example

```
IF (I .EQ. 0) THEN
              L=K+1
              K=I
              ELSE
              K=0
              ENDIF
```

4.8.0 DO-LOOPS

The general format for a DO loop is:

DO n i=m1,m2,m3

The DO statement works in the similar manner as the FOR, NEXT loop in BASIC. The DO Loop works as follows:

1) **i** is set to the value of **m1.**

2) After each pass through the loop (which ends with the statement labelled **n**), the step value, **m3** is added to **i.** If the **m3** term (step value), is omitted, then the step value is assumed to be one. Unlike other versions of FORTRAN the **i** and **m** terms do not have to be INTEGER values and the step may be negative. This allows fractional increments of the DO loop index, **i.** The ability of a negative increment, **m3,** allows loop to step in a downward direction. If the index is positive, the loop continues until the value of **i** is greater than that of **m1.** If the index is negative, the loop continues until the value of **i** is less than that of **m2. n** in the DO loop specifies the range of the DO loop. This is the statement number of the last statement of the DO loop.

Irrespective of the relation of the initial and ending values, the DO will **always** be executed once.

Note that 2 or more DO loops may end on the same statement.

DO loops may not terminate on **GO TO, STOP, IF-THEN-ELSE, END** or **RETURN** statements.

Example

```
DO 800 I=1,100
DO 1 J=I,END,.005
DO 99 A=START,END,AINCR
```

4.9.0 CONTINUE Statement

The format of the CONTINUE statement is:

CONTINUE

The CONTINUE statement is an executable FORTRAN statement. It generates no code in the object file, and is generally used as the terminal statement of a DO loop.

Example

```
     DO 100 I=1,50
        .
        .
        .
 100 CONTINUE
```

The CONTINUE statement does nothing. It simply serves to mark the range of the DO. It is also used for transfer of control i.e. you can GOTO it.

4.10.0 ERROR TRAPPING

Normally when an error occurs during the execution of a FORTRAN program a runtime error message will be produced. However using the **ERRSET** and **ERRCLR** statements it is possible to control and trap runtime errors.

The general format of these statements is:

ERRSET n,v
ERRCLR

where **n** is the label of the statement to transfer to if a runtime error occurs and **v** is the variable to contain the error code of the runtime error that occurred.

The ERRSET statement causes control to be transferred to the statement labeled **n** when a runtime error occurs. No runtime error message will be printed if the error is trapped with an ERRSET statement. The ERRSET statement can **only** be used if the **Q** option was specified on the $options statement. If an ERRSET or ERRCLR statement is encountered and the **Q** option was not specified, a compilation error will be generated.

The value placed in the variable **v** corresponds to the runtime error that occurred as follows:

1	Integer overflow
2	Convert error
3	Argument count error
4	Computed GOTO index out of range
5	Overflow
6	Division by zero
7	Square root of negative number
8	Log of negative number
9	Call stack push error
10	Call stack pop error
11	CHAIN/LOAD error
12	Illegal FORTRAN logical unit number
13	Unit already open
14	Disk full
15	Unit not open
16	Binary I/O to system console
17	Line too long on READ or WRITE
18	Format error
19	Input/Output error in READ or WRITE
20	Invalid character on input
21	Invalid input/output list (impossible)
22	Assigned GOTO error
23	CONTROL/C abort
24	Illegal character in input
25	File operation error
26	Seek error

If more than one ERRSET statement is executed in a routine, then the last one executed is the one in effect. If a runtime error should be trapped with an ERRSET statement, the ERRSET statement is automatically cleared after control has transferred to the statement **n.**

The ERRCLR statement clears the effect of the ERRSET statement in effect.

Example

```
$OPTIONS Q
     .
     ERRSET 10,CODE
     .
     .
     ERRCLR
     .
     .
     STOP
  10 TYPE 'ERROR ,ERROR CODE = ',CODE
     .
     END
```

4.11.0 CONTROL/C CONTROL

At the beginning of each READ or WRITE statement the state of the CONTROL/C abort flag is tested. If the CONTROL/C abort flag is set, then the console is tested to see if CONTROl/C has been hit. If CONTROL/C has been hit, then one of two actions will occur:

1) if there is an ERRSET in effect, the error branch will be taken with a **CONTRL/C** error.

2) otherwise a runtime error of **CONTRL/C** will occur.

The user has control of the CONTROL/C flag through the **CTRL ENABLE** and **CTRL DISABLE** statements. **CTRL ENABLE** set the CONTROL/C flag and allows a CONTROl/C from the console to abort the program. **CTRL DISABLE** resets the flag and causes the CONTROL/C to be ignored.

Example

```
      DO 1 I=1,100
      IF (I .EQ. 51)CTRL DISABLE
    1 TYPE I
      END
```

The above program will only abort if CONTROL/C is hit while the first 50 numbers are being output.

When program execution starts, the CONTROL/C flag will be set which allows CONTROL/C to abort the program.

Note: The CONTROL/C error, if enabled, can be trapped with an ERRSET statement. Also, the CIN function will return a control-C to the caller, regardless of the setting of the CONTROL/C flag

4.12.0 TRACING

There are two statements that are used to trace a program:

TRACE ON
TRACE OFF

When program execution begins tracing is initially **off** and must be explicitly turned on. Once tracing is on, it remains on until the program terminates or a **TRACE OFF** statement is executed. The effect of the trace statements is global over the whole program and tracing does not have to be turned on in each subroutine. The trace function will output the line number of the FORTRAN statement **before** execution only if the **X** option was specified on the options statement for this routine. Otherwise the program will be traced only up to the entrance to the subroutine. It should be noted that the line number for **any** entrance to a subroutine (either SUBROUTINE or FUNCTION) will always be output as **????** regardless of the state of the **X** option.

Example

```
      IF (FLAG .EQ. 0)TRACE ON
      TRACE OFF

      DO 1 I=1,100
      IF (I .EQ. 50)TRACE ON
1     TYPE I
```

4.13.0 DUMP statement

The general format of the DUMP statement is:

DUMP /ident/ output list

where **ident** is up to a 10 character identifier for this DUMP statement and **output list** is a standard WRITE output list that may contain variables, constants, character strings, array elements, array names and implied DO loops.

The DUMP statement is used to display information when a runtime error occurs that is **not** trapped by an ERRSET statement.

Example

```
DUMP /AFTER-DIVIDE/ 'Index after divide is ',K
```

More than one DUMP statement may be executed in a routine and the last one executed is the one that will be output on a runtime error. Each subprogram may contain its own DUMP statement, but only the last DUMP statement executed in a particular routine will be output.

5.0.0 Program Termination Statements

5.1.0 PAUSE Statement

The general format of the PAUSE statement is:

PAUSE 'any char string'

This statement causes the program to wait for any input from the system console. To continue execution, press any character on the system keyboard. If the character string option is specified, the string will be displayed on the system console. The string is enclosed in single quotes ('). To output a quote, two quotes in a row must be entered; e.g. ('') outputs as (''). The quotes surrounding the text are not output.

Example

```
PAUSE 'HIT ANY KEY TO CONTINUE'
PAUSE
PAUSE 'DATA OUT OF SEQUENCE, IGNORED'
PAUSE 'THIS IS A SINGLE QUOTE ('')   '
```

5.2.0 STOP Statement

The general format of the STOP statement is:

STOP 'any char string'
STOP n

When a STOP statement is encountered, it causes termination of the executing program. If the character string is specified it will be printed on the system console when the statement is executed. After the character string is output, the program terminates and returns to CP/M. The string is enclosed in single quotes. To output a quote, two quotes in a row must be entered. The quotes surrounding the text will not be output.

In the second form, **n** is a 1 to 5 digit integer number that will display. **n** is optional.

Example

```
STOP 'PROGRAM COMPLETE'
STOP 1267
STOP
STOP 'ERROR OCCURRED, CHECK OUTPUT'
```

5.3.0 END Statement

The format of the END statement is:

END

This is a required statement for every FORTRAN routine. It is used by the compiler to indicate the end of one logical routine. If an END statement is encountered during execution, then the message **END IN - XXXXX** will be output to the system console, with **XXXXX** being replaced by the name of the FORTRAN routine in which the END statement was executed.

6.0.0 Array Specification

An array is a collection of values that are referenced by the same name and the particular element is specified by a subscript. Subscripts can be real or integer expressions or constants and will be truncated to an integer value after the expression is evaluated.

An array may have from one to seven dimensions.

Example

If GRADE has 3 elements then:

```
GRADE(1)   refers to the first element
GRADE(2)   refers to the second element
GRADE(3)   refers to the third element
```

NOTE: Subscripted variables cannot be used as subscripts, thus GRADE(A(I)) is invalid, where both GRADE and A are arrays.

6.1.0 Dimension Statement

The general format for a DIMENSION statement is:

DIMENSION v(n1,n2,..,nm),..

The DIMENSION statement is used to define an array. The rules for using the DIMENSION statement are as follows:

1) Every subscripted variable must appear in a DIMENSION Statement whether explicit (in a dimension statement) or implied (in a REAL, INTEGER, LOGICAL or COMMON statement) prior to the use of the first executable statement.

2) A DIMENSION specification must contain the maximum dimensions for the array being defined.

3) The dimensions specified in the statement must be numeric in the main routine. However, in subroutines the subscripts may be integer variables. Hence the following statement is valid only in a subroutine:

```
DIMENSION A(I,J)
```

In the case where the dimensions of an array are specified as variables, the value of the variable at runtime will be used in computing the position within the array to be accessed.

4) All arrays passed to subprograms must be DIMENSIONED in the subprogram as well as in the main program. If the arguments in the subprogram differ from those in the main program, then only those sections of the array specified by the DIMENSION statement in the subprogram will be accessible in the subprogram.

5) The number of dimensions specified for a particular array cannot exceed 7.

6) No single array can exceed 32,767 bytes in size.

See the following for examples of the DIMENSION statement used both in the direct and indirect mode.

```
DIMENSION A(3,2,3),C(10),ZOT(10,10)
INTEGER SWITCH(15)

"A" would require 3*2*3*(6) = 108 bytes
"C" would require 10*(6) = 60 bytes
"ZOT" would require 10*10*(6) = 600 bytes
"SWITCH" would require 15*(6) = 90 bytes
```

WARNING: No subscript range checking is performed at runtime.

Example

```
DIMENSION GUN(S,E)
DIMENSION A(2,2),B(10)
DIMENSION ZIT(10)
REAL APPLE(10)
LOGICAL FUNCT(100)
```

In calculating the memory used by an array, multiply each of the dimensions times each other, then times 6. The result will be the number of bytes used by the array for storage. In the above examples of the dimension statement, the arrays would require the following storage.

6.2.0 Subscripts

Subscripts are used to specify an entry into an array (i.e. the value specified in the subscript is the element of the array referenced). Subscripts may be integers, real (fractions are truncated), logical expressions or any other valid expression. Expressions are evaluated as explained in the EXPRESSION section (3.2.0).

Example

```
ZIT(8)
A(1+2)
ORANGES(I+5-(K*10)/2)
APPLE(5)
```

7.0.0 Subprograms

Subprograms provide a means to define often needed sections of code that can be considered as a unit. FORTRAN provides the means to execute these subprograms whenever they are referenced.

There are 3 types of subprograms supported in this version of FORTRAN:

1) SUBROUTINE subprograms

2) FUNCTION subprograms

3) Built in library functions

The major differences between FUNCTIONS and SUBROUTINES are listed below.

1) FUNCTIONS are used in expressions, while SUBROUTINES must be CALLed.

2) FUNCTIONS require at **least one** parameter; SUBROUTINES do not require any.

3) The name on the FUNCTION statement must be the object of a replacement statement somewhere in the FUNCTION; this is not the case for a SUBROUTINE.

WARNING: If a constant is passed as an argument in either a CALL or FUNCTION reference, and the corresponding parameter in the SUBROUTINE or FUNCTION is modified, then the value of the constant that was passed will be changed, and remain that of the new value.

NOTE: All SUBROUTINES and FUNCTIONS **must** be compiled at the same time. Also SUBROUTINE and FUNCTION names are only significant to the first 5 characters.

7.1.0 SUBROUTINE Statement

The general format of the SUBROUTINE statement is:

SUBROUTINE name(list)

The SUBROUTINE statement is required to identify the beginning of a logical routine. This statement, or one of similar function is required at the beginning of every SUBROUTINE. The **list** that is to receive the values being passed to the subroutine is optional if no parameters are to be passed.

Example

```
SUBROUTINE ADDIT (RESULT,X,Y)
RESULT=X+Y
RETURN
END
```

7.2.0 FUNCTION Statement

The general format of the FUNCTION statement is:

FUNCTION name(list)

A FUNCTION Statement is used to define a logical section of code as a FUNCTION. The type of result of a FUNCTION can be specified by preceding the FUNCTION with REAL, INTEGER or LOGICAL or the name of the FUNCTION may appear in a type statement within the FUNCTION.

Example

```
FUNCTION SWAP(A)
SWAP=A
RETURN
END

INTEGER FUNCTION SWAP (A)
SWAP=IFIX(A)
RETURN
END

FUNCTION SWAP (A)
INTEGER SWAP
SWAP=A/2
RETURN
END
```

7.3.0 CALL Statement

The general format of the CALL statement is:

CALL name(list)

The CALL statement is used to transfer control to a SUBROUTINE. **List** specifies the parameters to be passed to the SUBROUTINE and may be omitted if no parameters are to be passed.

The number of parameters in a CALL and SUBROUTINE statement referring to the same subprogram **must** be the same, otherwise a runtime error will result.

Example

```
CALL XSWAP (NUM1,NUM2,TOTAL)
CALL XSWAP
```

7.4.0 RETURN Statement

There are 2 types of RETURN statements:

1) normal RETURN

2) multiple RETURN

7.4.1 Normal return

RETURN

The RETURN Statement is used to terminate execution of a subprogram whether it is a FUNCTION or a SUBROUTINE. Return is transferred to the next statement following the CALL statement, or in the case of a FUNCTION, return is transferred back to the point where it was called with the value of the FUNCTION returned. A RETURN statement is not valid in the MAIN routine and will cause an error during compilation.

Example

```
SUBROUTINE ZERO(I,J)
I=0
J=0
RETURN
END
```

7.4.2 Multiple return

The general format of the multiple RETURN statement is:

RETURN I

This variation of the RETURN statement is used to transfer back from a SUBROUTINE to a point other than the statement that immediately follows the CALL. The **I** in the RETURN is the name of a variable in the argument list of the subroutine and must have been passed as a label in the CALL. The CALL statement that invokes a routine that contains a multiple return must pass the label as one of the parameters. The statement label is indicated in the argument list by preceding the label with an ampersand (&).

Example

```
        CALL X(&1,Y,2,&2)
        .
        .
        SUBROUTINE X(I,A,IC,J)
        .
C
C THE FOLLOWING RETURN WILL TRANSFER TO THE STATEMENT
C LABELLED '1' IN THE CALLING PROGRAM.
C
        RETURN I
        .
C
C THE FOLLOWING RETURN WILL TRANSFER TO THE STATEMENT
C LABELLED '2' IN THE CALLING PROGRAM.
C
        RETURN J
        END
```

NOTE: Multiple RETURNS are only valid for SUBROUTINES.

7.5.0 BLOCK DATA SUBPROGRAM

The BLOCK DATA subprogram is used to initialize variables in **named** COMMON. The BLOCK DATA subprogram must contain **no** executable statements but may contain only declaration statements for specifing variable types, array dimensions, COMMON blocks and DATA statements.

Example

```
BLOCK DATA
INTEGER FIRST,LAST
COMMON /ONE/ NAMES(100) /TWO/ FIRST,LAST
DATA FIRST /1/, LAST/10/
DATA NAMES /1,2.0,4,5,6,7,8,9,10,90*99999/
END
```

It should be noted that the variable in named COMMON can be initialized in any routine and the BLOCK DATA subprogram appears only for compatibility with other FORTRAN systems.

8.0.0 Input/Output

8.1.0 Introduction to FORTRAN I/O

8.1.1 General Information

Input and Output (I/O) under FORTRAN may take one of the following forms:

1) Standard Formatted I/O
2) Free Format I/O
3) Binary I/O

In formatted I/O, input and output is defined in terms of fields which are right justified on the decimal point, with zero suppression. In a FORMAT statement, no more than three levels of nested parenthesis are allowed (outer set and two nested inner sets).

Free Format I/O is used as in BASIC. All the values are entered using commas (,) or carriage returns to delimit the numbers.

Binary I/O is a third option that allows passing of large files between FORTRAN programs, with the minimal amount of wasted disk space. Each variable written in binary format uses six bytes of disk space.

FORTRAN logical units 0 and 1 are dedicated to console input and output and cannot be either opened or closed. An attempt to open or close 0 or 1 will result in a runtime error. Logical unit 0 is used for console input and logical unit 1 is used for console output. Binary I/O cannot be specified for logical units 0 or 1 and doing so will result in a runtime error.

There are two special I/O statements:

TYPE
ACCEPT

Both of these are followed by a standard I/O list. TYPE is equivalent to WRITE (1,*) and ACCEPT to READ (0,*). This is just a convenient method of doing console I/O.

Example

```
TYPE I,J,(A(I),I=1,10)
ACCEPT 'INPUT THE MAX COUNT',COUNT
```

A RUNTIME format can be specified for any formatted I/O statement by substituting an **ARRAY** name for the FORMAT number. At runtime, the array is assumed to contain a valid FORTRAN format (complete with its outer set of parentheses). This allows a FORMAT statement to be input at runtime and then to be used. The format should be input using an A6 format specification as **imbedded** blanks (added in using less than an A6) will cause a runtime error.

Example

```
      DIMENSION FORM(10)
      READ (0,10) 'ENTER DATA FORMAT ',FORM
   10 FORMAT (10A6)
      .
      .
      .
      READ (4,FORM) A,B,C
      .
      .
      .
      WRITE (10,FORM) R1,R2,R3
```

8.1.1 I/O List Specification

The I/O List is used to specify which variables are to be READ or WRITTEN in a particular I/O statement. The list has the same form for both READ and WRITE statements. The list can be composed of one or more of the following:

1) simple (non-subscripted) variable
2) array element
3) array name
4) implied DO loop
5) literal
6) constant (WRITE only)

The above types are combined to form the I/O list specification. Items 1-4 are self explanatory, however item 4, the implied DO loop is explained further below:

The implied DO loop is used mainly to output sections of one or more arrays and functions in the same way as does a regular DO loop. An example of an implied DO loop is:

```
WRITE (1,*) (F(I),I=1,3,1)
```

It should be noted that the outer parentheses and the comma preceding the DO index are always necessary when using an implied DO loop. Nested loops can be used. Each loop must be enclosed in parentheses. An example follows:

```
WRITE (1,*) (J,(F(I,J),I=1,4),J=1,30,2)
```

The inner DO (I) is performed for each iteration of the outer DO (J). Note that other than array elements can be included within the range of an implied DO. Implied DO's can be nested to any depth, each within its own set of parenthesis.

LITERALS (character strings enclosed in quotes) can be used in any WRITE statement and in READ statements that **reference** the system console. The literals can be used as prompts for input or identification on output.

Example

```
WRITE (1,*) 'A= ',A
TYPE 'The answer is ',ANS
WRITE (10,3) 'X= ',X

READ (0,*) 'A= ',A,' B= ',B
ACCEPT 'Enter quantity ',QUANT
```

An attempt to use a literal in a READ statement that doesn't reference the console will result in an **INPUT ERR** runtime error.

8.2.0 READ Statement

READ(unit{,format}{,END=end}{,ERR=error}) I/O list

The READ Statement is required in order for the user to do input through the FORTRAN system. If a **unit** number of **0** is used there is no need to open this file as it is assumed to be system console input. Note: do not use **1** as the logical **unit** as it is reserved for the system console output. Any other **unit** number must first have been opened by the user through the OPEN or LOPEN subroutine. The FORMAT entry may take one of the following forms:

1) The **FORMAT** number is the label on the FORMAT statement which is to be used.

2) An asterisk (*) indicates that input is to be free format. The exact format of the output depends on the value of the number being output and is determined at runtime.

3) If this entry is left blank (or not specified), binary input is assumed.

4) The name of an array that contains the format to be used.

END= is the label to which transfer of control is to be made should an end of file condition be encountered. **ERR=** is the label to which control will be transferred, should an error other than end of file occur during input. **I/O list** is the string of variables which accept the data to be read.

Example

READ (0,2) A	read from the system console the variable 'A' under format number 2
READ (0,*) A	read from the system console the variable 'A' in free format.
READ (4) A	read from logical file 4, the variable A in binary.
READ (4,,END=10) A	read from logical file 4, the variable A, in binary, and if end-of-file is encountered, go to statement label 10
READ (4,*,END=10,ERR=100) A	read from logical file 4, the variable A in free format, should end-of-file be encountered, go to statement label 10. If an error occurs, go to statement label 100.
READ (4,,ERR=100) A	read from logical file 4, the variable A in binary format, and if an error occurs, go to statement label 100.

The **END=** and **ERR=** parameters are optional and can appear in any order.

8.3.0 WRITE Statement

WRITE (unit{,format}{,END=end}{,ERR=error}) I/O list

The WRITE statement is the opposite of the READ statement in that it converts the values of the variables specified to a format that is understandable to the user. The **I/O list** is specified exactly the same as for the READ statement with the exception that a string can always be used in the **I/O list.** However the **END=** serves no function and will never be used by the WRITE statement.

Example

```
WRITE (1,2) I,J,PAY,WITHOLD
WRITE (1) (I,I=1,10)
WRITE (10,*) THIS
WRITE (6,12,END=99,ERR=66) LOOP,COUNT
```

8.4.0 MEMORY TO MEMORY I/O statements

The ENCODE and DECODE statements allow I/O to be performed to or from a specified memory location. This allows data in memory to be read (using DECODE) with perhaps a different format code depending on the data itself. The ENCODE statement is similar to a WRITE statement in that data is formatted according to the specified format type, but instead of being output to a file it will be placed in memory at the specified location for further processing.

8.4.1 DECODE statement

The general form of the DECODE statement is:

DECODE (variable,length,format) I/O list

The DECODE statement is similar to a READ statement in that it causes data to be converted from external ASCII format to internal FORTRAN type. **Variable** is either an unsubscripted variable name or an array name. **Length** is the number of bytes to process for this READ starting at **variable.** If multiple records are required by the I/O list, successive records of **length** will be retrieved from memory. Input records will be blank padded on the right end as necessary as in a READ statement. **Format** is either an asterisk for free formatting or the number of a FORMAT statement.

Example

```
     DIMENSION A(15)
     READ (1,10) A
10   FORMAT (15A6)
     DECODE (A,80,11) KNT1,KNT2,CNT3
11   FORMAT (I10,I3,F10.5)
```

8.4.2 ENCODE statement

The general form of the ENCODE statement is:

ENCODE (variable,length,format) I/O list

The ENCODE statement is similar to a WRITE statement in that it is used for a memory to memory formatted WRITE. **Variable** is either an unsubscripted variable name or an array name. **Length** is the number of bytes (or characters) that the output record is to contain. If the number of characters generated by the ENCODE statement is less than **length**, then the record will be blank padded to length. If the number of characters in the generated record is greater than **length**, then the record will be truncated after the **length** character. If multiple output records are generated, successive records of **length** character will be placed in memory starting at **variable. Format** is either an asterisk for free formatting or the number of a FORMAT statement.

Example

```
DIMENSION A(15)
ENCODE (A,80,*) (I,I=1,5)
```

8.5.0 Format Statement and Format Specifications

The general form of the FORMAT statement is:

n FORMAT (s1,s2,...sn)

The FORMAT Statement is used in FORTRAN to do formatted input and output. Through the use of this statement the programmer has the ability to select the fields in which to read, or specify the columns on which to write. It is the use of this statement which gives FORTRAN its I/O power. On FORMATTED input, blanks are treated as if they were zeroes except when reading in A format. A constant enclosed in **backslashes** can be used to enter a binary constant from a string within a FORMAT statement.

If a number cannot be written in the specified field width, then the entire field will be filled with asterisks (*) to indicate the error condition. Note: some FORTRANS will print a negative number, even when there is not room enough to place the negative sign in the field by omitting the negative sign. In this case, NEVADA FORTRAN will asterisk fill the field. The asterisk filling of a field that is not large enough to output a number applies on all output specifications. A **ZERO** will always be printed as **0.0** under a **F** or **E** field specification. If a field is printed as **0.000...** this indicates that the digits have been truncated because the **d** portion of the field specification was not large enough.

All floating numbers output using the **F** and **E** (and **G** with a floating point number) specifications will be **rounded** to the appropriate number of digits specified by the **d** portion of the field specifier.

8.5.1 X-Type (wX)

The X-Type specification is used to space over any number of columns with a maximum of 255 character positions. **w** may have any value from 1 to 255.

On output the columns spaced over will be set to blanks. On input **w** characters of the input record will be skipped.

Example

```
10 FORMAT (10X,I10)
```

8.5.2 I-Type (Iw)

The I-Type specification is used as a method of performing I/O with integer numbers. On input, the number must be right justified in the specified field with leading zeros or blanks. On output the leading zeroes are replaced by blanks, and the number is right justified in the field.

Example

```
10 FORMAT (10I10)
```

8.5.0 Format Statement and Format Specifications

The general form of the FORMAT statement is:

n FORMAT (s1,s2,...sn)

The FORMAT Statement is used in FORTRAN to do formatted input and output. Through the use of this statement the programmer has the ability to select the fields in which to read, or specify the columns on which to write. It is the use of this statement which gives FORTRAN its I/O power. On FORMATTED input, blanks are treated as if they were zeroes except when reading in A format. A constant enclosed in **backslashes** can be used to enter a binary constant from a string within a FORMAT statement.

If a number cannot be written in the specified field width, then the entire field will be filled with asterisks (*) to indicate the error condition. Note: some FORTRANS will print a negative number, even when there is not room enough to place the negative sign in the field by omitting the negative sign. In this case, NEVADA FORTRAN will asterisk fill the field. The asterisk filling of a field that is not large enough to output a number applies on all output specifications. A **ZERO** will always be printed as **0.0** under a **F** or **E** field specification. If a field is printed as **0.000...** this indicates that the digits have been truncated because the **d** portion of the field specification was not large enough.

All floating numbers output using the **F** and **E** (and **G** with a floating point number) specifications will be **rounded** to the appropriate number of digits specified by the **d** portion of the field specifier.

8.5.1 X-Type (wX)

The X-Type specification is used to space over any number of columns with a maximum of 255 character positions. **w** may have any value from 1 to 255.

On output the columns spaced over will be set to blanks. On input **w** characters of the input record will be skipped.

Example

```
10 FORMAT (10X,I10)
```

8.5.2 I-Type (Iw)

The I-Type specification is used as a method of performing I/O with integer numbers. On input, the number must be right justified in the specified field with leading zeros or blanks. On output the leading zeroes are replaced by blanks, and the number is right justified in the field.

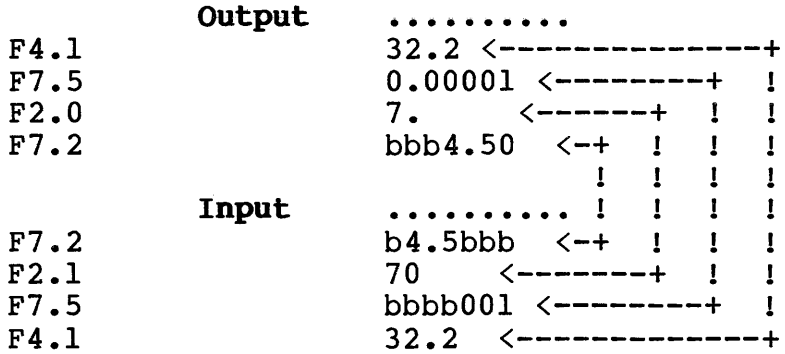
Example

```
10 FORMAT (10I10)
```

8.5.3 F-Type (Fw.d)

The F-Type specification is one of several specifications for performing I/O with floating point numbers. The digit portion of the decimal number works the same as in the I-Type format. The fractional part of the number is always printed, including trailing zeroes. During input, the decimal point is assumed to be at the indicated position, unless explicitly overridden in the input field. The number ZERO will always print as **0.0** (with the decimal point aligned where specified) regardless of the field width or decimal digits specified. Remember to consider the decimal point and negative sign of the number when specifing the width of the output field.

Example



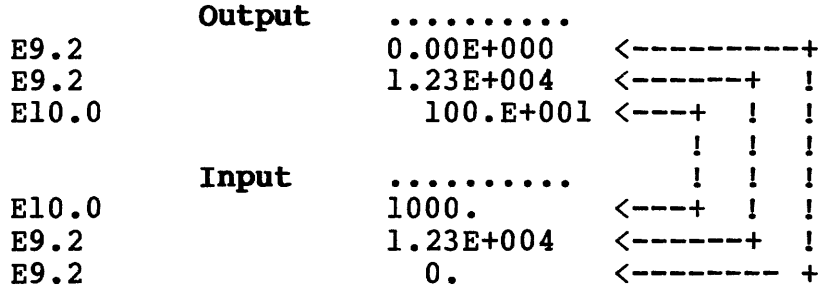
NOTE: b is used to indicate a blank position.

During input the F field specifier reads **w** characters. If there is **not** a decimal point in the field read, a decimal is inserted **d** digits from the right. A decimal point in the input field overrides the field specification.

8.5.4 E-Type (Ew.d)

The E-Type specification is another method of performing I/O with floating point (real) numbers. It is through this specification that the programmer may perform I/O using an exponential format. That is a mantissa followed by an exponent of ten. Again as with the F type, the decimal point is assumed to be at the indicated position if not overridden in the input field. The exponent part of the input number can be omitted, in which case it is treated as if it were an F type specification. The number will be printed as **d** digits followed by the letter E, exponent sign, and a three digits exponent.

Example



Data can be read in the **F** format using the **E** format specification without causing an error.

8.5.5 A-Type (Aw)

The A-Type specification is used to perform the input of alphanumeric data in ASCII character form. Up to 6 ASCII characters may be stored per variable name. However, this is entirely under program control. For example the user may choose to store only one character per variable in a dimensioned array, in order to do character manipulation. Characters are stored in the variable left justified and zero filled. On output these padding zeroes will be printed as blanks. It is not advisable to perform any arithmetic operations on a variable that contains character data because unpredicatable results may occur.

Example

```
10 FORMAT (A10,I10,A6)
```

8.5.6 /-TYPE (/)

The /-Type specification is used to cause I/O to skip to the next record. During input this causes a new input record to be read, even though the previous one was not fully used. On output the slash will cause the current line to be written out to the associated file.

Example

```
     54 FORMAT (I10/)

     WRITE (1,100) 1,20,45
100  FORMAT (I3/2I3)

     will generate

  1
 20 45
```

8.5.7 Z-Type

The Z-Type specification is used only for output, to indicate to the system that a carriage return/line feed is **not** to be written at the end of the record. The Z specification is ignored on input.

Example

```
     WRITE (1,10)
10   FORMAT ('INPUT X ',Z)
     READ (0,*) X
```

8.5.8 L-Type (Lw)

The L-Type specification is used with LOGICAL variables, where **w** is the width of the field. On output, the letter T or F is printed (for TRUE or FALSE respectively). The T or F will be right justified in the field. On input, the field is scanned from left to right until a T or F is found (again for TRUE or FALSE). The T or F can be located anywhere in the field and all characters that follow the T or F in the remainder of the field are ignored. If the first character found is not a T or F an error will be generated. If the input field is completely blank, then a FALSE value will be used.

Example

```
      LOGICAL WHICH
      WRITE (1,11) WHICH
11    FORMAT (8L10)
```

8.5.9 T-Type (Tw)

The T-Type code can be used on both input and output. It is used to move to a explicit column within the input or output buffer. **W** specifies an absolute column number that the next character is to be read from (on input) or to be placed to (on output). The first column number is 1. On input the T format code can be used to re-read a particular set of columns in different format codes in the same read statement. Tabbing beyond the end of the input record causes the input record to be blank padded. On output, the output cursor can be moved back (to the left) over text already inserted into the output buffer, thus causing text and ready there to be over written with new data. Tabbing beyond the maximum character inserted into the output buffer will cause blanks to be inserted into the output buffer to the indicated column. The maximum value of **w** is 255.

Example

```
      WRITE (1,56) I,LOT
56    FORMAT (I10,T50,I4)
```

8.5.10 K-Type (Kw)

The K-Type format code is used to transmit data in hexadecimal format. Each byte of internal memory occupies 2 hexadecimal characters. If **w** is less than the 12 (6 bytes/variable, 2 hex characters/byte), the hexadecimal characters will be either input or output starting from the low order memory address (beginning of the variable).

Example

```
     WRITE (1,99) 1
99   FORMAT (K12)
```

will output the line:

100000000081

8.5.11 G-Type (Gw.d)

The G-Type can be used on either input or output and for both integer and real values. The G format is treated as follows:

Output

If the output element is of type integer, then the format code used will be Iw.

If the output element is of type real, the actual format code used depends on the value of the number being output:

Ew.d will be used if the number is outside the range of

```
F(w-5).d,5X        if      .1 <= number <1
F(w-5).(d-1),5X    if       1 <= number <10
                   .
                   .
                   .
F(w-5).1,5X        if       10**(d-2)<= number< 10**(d-1)
F(w-5).0,5X        if       10**(d-1)<= number< 10**d
```

In general in this range:

F(w-5).(d-(exponent of number)),5X

Input

If the input element is of type integer: Iw
If the input element is of type real: Ew.d

Example

```
     A=5.67
     WRITE (1,34) A
34   FORMAT (G10.5)

     READ (0,9) A
9    FORMAT (G9.3)
```

8.5.12 Repeating field specifications

A field specification can be repeated in a FORMAT statement by preceding it with the number of times that it should be repeated. Thus 4I10 is the same as I10,I10,I10,I10. The following FORMATS are equivalent:

```
10   FORMAT (3I4,3F10.4)
10   FORMAT (I4,I4,I4,F10.4,F10.4)
```

A single field specification or a group of field specifications can be enclosed in parentheses and preceded by a group count. In this case, the entire group is repeated the specified number of times. The following FORMATS are equivalent:

```
19   FORMAT (I4,2(I3,F4.1))
19   FORMAT (I4,I3,F4.1,I3,F4.1)
```

The FORMATS:

```
10   FORMAT (I5,2(I3,F5.1))
10   FORMAT (I5,I3,F5.1,I3,F5.1)
```

execute exactly the same for output, but differ for input. In a FORMAT without group counts, control goes to the beginning of the FORMAT statement for read or writing of additional values. In a FORMAT with group counts, additional values are read according to the last complete group.

Example

```
     READ (2,10) KNT,(Z(I),I=1,KNT)
10   FORMAT (I5/(F10.5))
```

The I5 specification will be used once and the array values will be read using the F10.5 specification.

Group counts can be nested to a maximum depth of two. Thus:

```
10   FORMAT (2(I5,3(I10))          is ok, while
10   FORMAT (2(I5,3(I10,2(I1)))    is not legal.
```

8.5.13 String Output

Character strings are written using a FORMATTED write. The string to be written is enclosed in SINGLE QUOTES (') and may not contain a backslash. To output a single quote within the string, two single quotes in a row must be entered. The string format type is only valid on output and if used with a READ will result in a runtime error being produced.

A character string can also be specified using the **H** (or Hollerith) field specification. This is an awkward method of specifing a character string as the number of characters in the string must be specified in front of the **H**. The **H** type should be avoided as it can lead to problems.

The hexadecimal code for any character (**except 0**) can be inserted in a string by enclosing it in backslashes (\). The backslash character can be changed using the CONFIG program.

Placing an ampersand in front of a character in a string causes the character to be treated as a control character. To output an ampersand, **two** ampersands in a row must be used.

Example

```
     WRITE (1,46)
46   FORMAT ('THIS IS A TEXT STRING')
65   FORMAT (21HTHIS IS A TEXT STRING)
48   FORMAT ('This is an exclamation point\21\')

          generates:     This is an exclamation point!

99   FORMAT ('This is a control L: &L')

          generates:     This is a control L: (followed by a
                         control/L

11   FORMAT ('This is an ampersand: &&')

          generates:     This is an ampersand: &
```

8.6.0 Free Format I/O
8.6.1 INPUT

FREE format input is similar to BASIC. Blanks in this mode of input are ignored completely. Numbers are entered in any format (F, I or E) and can be intermixed as desired. Numbers must be separated from each other by a comma or a carriage return. A comma may appear after the last number on an input line and is ignored if present. If the I/O list specifies more variables than there are in an input record, succeeding records will be read until the list is satisfied. Blank input records and blanks imbedded in numbers are ignored in this mode. The last number in any input record does not have to be followed by a comma.

8.6.2 OUTPUT

The type of output format used depends on the type of the variable or constant being output. An integer will result in an I type format being used, and a real will use a G type. (The actual format used in this case depends on the value being output).

8.7.0 BINARY I/O

BINARY I/O provides a quick and efficient means of transferring information to and from a file. The variables are READ or WRITTEN in BINARY format. That is, six bytes for each item in the I/O list. WRITE causes the item in the I/O list to be written exactly as it is stored in memory without any additional conversion. READ does the opposite, reading six bytes directly into the I/O list item. No conversion or check is made on the data being read.

Example

```
WRITE (1) (I,I=1,100)
WRITE (1,,ERR=66) ARRAY

READ (1,,END=99) VALUE
READ (1) THIS,IS,IT
```

Warning: the binary READ and WRITE transfers 6 bytes from the file specified directly to the variable in the I/O list. No check on the validity of the data is performed and the user should be sure that the variable contains valid numerical data before any arithmetic operations are done on the variable. An end-of-file is indicated by either the physical end of the file or a six byte field of all FF (hex). This is the value that ENDFILE will place at the end of a file that has had binary writes performed on it.

8.8.0 REWIND Statement

The general format of the REWIND statement is:

REWIND unit

The REWIND Statement is used to position the file associated with **unit** to the beginning of the file. Essentially this statement closes and then re-opens the file at the beginning.

8.9.0 BACKSPACE Statement

The general format of the BACKSPACE statement is:

BACKSPACE unit{,error}

The BACKSPACE statement is used to backspace **unit** one record. **Error** will contain the error code if the BACKSPACE fails. The BACKSPACE statement is currently **not** implemented and will produce a message to that effect if encountered at runtime.

8.10.0 ENDFILE Statement

The general format of the ENDFILE statement is:

ENDFILE unit

The ENDFILE statement is used to force an end-of-file on **unit.** Any data that existed beyond the point in the file where the ENDFILE was executed will be lost.

8.11.0 GENERAL COMMENTS ON FORTRAN I/O UNDER CP/M

The OPEN subroutine is used to associate a file with a FORTRAN logical unit. Eight files are available, numbered 0 through 7 with 0 being permanently open and associated with input from the CP/M console, logical file 1 also is permanently open and is associated with output to the CP/M console. Logical files 0 and 1 cannot be opened or closed. Additionally any logical unit associated with the CP/M console (through the use of the filename CON:) cannot have binary I/O done to it, cannot be rewound (using REWIND), endfiled (using ENDFILE) or seeked on (using the SEEK routine).

A file that is going to be written on should be deleted, using the DELETE subroutine, before the file is opened. The OPEN routine does not delete a file as it does not know what type of I/O will be performed on it.

The CLOSE routine will **not** place any end-of-file indicator in a file that was written to; the ENDFILE statement must be used to write an end-of-file indicator to a file. The ENDFILE statement will write the normal CP/M end-of-file indicator (control-Z) if the file specified in the ENDFILE has been written to and **no** binary I/O was done to the file. If binary I/O has been done to the file, then an end-of-file of 6 bytes of FF (hex) will be written instead. If a file is written and then read without being ENDFILEd, it is possible to encounter unwritten data of unknown characters that may cause an error during the READ (illegal character, end-of-file, etc). All files that are written to should be ENDFILEd.

When SEEKing on a file, remember that it is a **BYTE** position that is specified in the call to SEEK. Each record written to a file will contain a carriage return and line feed appended to the end of it. Remember that the carriage return and line feed **MUST** be included in the count of characters that make up a record. If SEEKing on a record basis, it is up the programmer to insure that each record written contains the same number of characters. If the records do not contain the same number, SEEKing can become a very complicated task. Calling the SEEK routine with a negative byte position will result in the file being positioned to its end-of-file position. The file can then be extended by ordinary WRITE statements. A READ after a position to end-of-file results in an immediate end-of-file condition being returned.

8.12.0 SPECIAL CHARACTERS DURING CONSOLE I/O

Entering a control-X during input from the CP/M console will cancel the current line and echo an exclamation point (!) followed by a carriage return and line feed.

End-of-file from the CP/M console is indicated by a control-Z being entered as the first character of an input line during console I/O.

Entering a DELETE (7F hex) or control-H will erase the last character entered.

9.0.0 OPERATION

9.1.0 Getting Started

Hardware required

1. 8080/8085/Z80 processor
2. Minimum of 48K of RAM for the compiler
3. At least one disk drive

Software Required

1. CP/M
2. Any text editor

Files on the Distribution Disk

FORT.COM is the FORTRAN compiler
FRUN.COM is the runtime execution package
FORT.ERR is the compiler error text file.
CONFIG.COM is a program to generate the error file and setup compiler and runtime defaults.
ERRORS is the error text file used by **CONFIG.**
(Also see the Nevada Assembler Manual for other files not listed here)

GETTING STARTED

The very **first** thing that you should do is to make at least one backup copy of your Nevada FORTRAN diskette. The original diskette should be kept in a safe place in case its ever needed in the future.

You can use your systems disk copy program or PIP to make copies of the original Nevada FORTRAN diskette.

On 5 1/4" diskettes you may have to remove (erase) other programs to make room for all the Nevada FORTRAN programs, before the next step. Make sure the default drive has at least 8K of available space. If it does not, you will get a BDOS write error - CP/M's way of letting you know the disk is full!. In some cases you will have to temporarily remove some of the FORTRAN example programs to make the space available.

NOTE: The Nevada assember ASSM.COM **must** be on the same disk which contains the FORTRAN compiler

Generating the compiler error file FORT.ERR

The program **CONFIG** reads the text file **ERRORS** which contains all the compiler error messages. These messages may be changed with the restriction that they can only be one line, the first 2 characters are the error number followed by a blank. To generate the error file, just enter

CONFIG at the CP/M prompt and reply **Y** to the question about generating the error file. You **must** generate the error file as it is not supplied on the disk. This only needs to be done once or whenever any of the error text is changed.

Certain default compiler and runtime parameters can be set by the same program. Just enter carriage return (or ENTER) to leave the default as it is, or enter the new default value. The character used to delimit hexadecimal constants in strings can also be changed. Also the method that the runtime package performs CP/M console I/O can be specified as either CP/M function 1&2, CP/M function 6 (for 2.0 only) or direct BIOS calls. Specifing CP/M functions 1&2 allows you to use control-P to send a copy of your FORTRAN output to the printer.

Since NEVADA FORTRAN supports the North Star floating point board, you must specify the address of the board if you have one and want FORTRAN to use it (see section 11.9.0).

After creating FORT.ERR you can erase CONFIG.COM and ERRORS if you need the disk space.

Creating a Program

A program can be created with any of the numerous available text editors. The name of the FORTRAN source program should have the filename extension of **.FOR**, such as PROG.FOR. Refer to section 1.3.0 for a detailed description of the format of each source statement.

9.2.0 COMPILING A PROGRAM

The general format of the command to compile a FORTRAN program is:

FORT U:PGM.LAO $OPTIONS

where:

FORT is FORT.COM, the FORTRAN compiler

PGM is the FORTRAN source program to compile and has the extension **.FOR.**

U: is the drive where PGM.FOR is located (if not present, the default drive is used).

L is the drive for the listing as follows:

A-P	uses that drive for the listing
X	listing to CP/M console
Y	listing to CP/M lst: device
Z	do not generate a listing

The listing will have the same filename as the source file but with the extension **.LST**.

A is the drive for the intermediate assembly file as follows:

A-P uses that drive
Z do not generate an assembly file.

The assembly file will have the same filename as the source file but with the extension .ASM. This file is normally deleted by the FORTRAN compiler.

O is the drive for the final object program as follows:

A-P uses that drive
Z don't generate an object file

The object program will have the same filename as the source file but with the extension **.OBJ**.

Notes:

If **Z** if specified in either the assembly or object drive position, no object program will be generated.

If the three drive specifiers are not specified, then the default drive will be used.

Both **FORT.COM** and **FORT.ERR** (the error file) must be present on the default drive.

If the **O** is not specified as **Z**, then the assembly file will be automatically assembled and the intermediate .ASM file will be deleted.

9.3.0 COMPILE OPTIONS

Options that effect the compilation of the FORTRAN program can be specified on the command line by preceding the option string with dollar sign ($). The following options can be specified:

N

No assembly file will be produced (and no object file also).

P

The listing file (if specified) will be paginated (66 lines to a page). Each new FORTRAN routine will start on a new page.

1

Source statements will be blank padded to 64 characters.

2

Source statements will be blank padded to 72 characters.

NOTE: Normally source statement are **not** blank padded. This may cause a problem where blanks are wanted inside a literal string and the string is started on one statement and continued over one or more continuation statements. Without the pad option, the trailing blanks may be lost (of course you could break the continued literal into several, making sure that there is a quote after any blanks at the end of a statement). For example:

```
     WRITE (1,10)
10   FORMAT ('THIS IS
    *A TEST')
```

Produces: THIS IS A TEST
without blank padding and

THIS IS A TEST
with blank padding.

This could be written as:

```
     WRITE (1,10)
10   FORMAT ('THIS IS                    ',
    *'A TEST')
```

to produce the same results as with the blank padding specified.

H

This option is used to conjunction with the **P** option to suppress the heading in the listing.

C=XXXX

This option specifies the maximum number of COMMON blocks that may be defined in the program to be compiled. The default is 15.

B=XXXX

This option specifies the size of the input statement buffer. The default is 530 characters and the buffer **must** be large enough to contain a complete statement (first record plus all continuations).

M=XXXX

This option specifies the memory address at which **blank** COMMON will **end.** In other words, blank COMMON will be allocated **downward** in memory from the specified address. The address specified must be in hexadecimal.

This option is useful in forcing blank COMMON to be allocated at the same address in memory for passing data between routines that **CHAIN** to each other.

Examples

```
FORT MYPROG $C=20
```

Compiles MYPROG.FOR from the default drive, generating MYPROG.ASM, MYPROG.LST and MYPROG.OBJ on the default drive and allowing for the definition of up to 20 COMMON blocks.

```
FORT B:READ.XCD $P2
```

Compiles READ.FOR from drive B, generating READ.ASM on drive C and the listing to the console. The listing will be paginated and source statements will be padded to 72 characters.

```
FORT TEST.YZZ $P
```

Compiles TEST.FOR from the default drive, no .ASM or .OBJ file will be produced but a paginated listing will go to the CP/M list (LST:) device.

```
FORT UPDATE.XBB $PH
```

Compiles UPDATE.FOR from the default drive, generating UPDATE.ASM on drive B, a paginated listing minus the heading line to the console and no .OBJ file.

9.4.0 EXECUTING A PROGRAM

Once the object file has been produced, the program can be executed by simply typing:

FRUN u:filename

where **u:** is optional and if not present, the default drive is used. The FORTRAN runtime package, **FRUN** occupies memory from 100H to 3FFFH. It will load the program to be executed starting at 4000H. The program is then executed and continues until either it terminates normally or a runtime error occurs.

10.0.0 General Purpose SUBROUTINE/FUNCTION Library

The following list of subroutines are available for the user of FORTRAN.

SUBROUTINE Name

OPEN
LOPEN
CLOSE
CREATE
KILL
SEEK
RENAME
SETUNT
MOVE
CHAIN
LOAD
EXIT
DELAY
CIN
CTEST
OUT
SETIO

FUNCTION Name

INP
CALL
CBTOF

All SUBROUTINES are accessed through the CALL statement described previously. For details as to the parameters required, see the following descriptions of the individual routines. If **error** is present and a CP/M error should occur, return will be to the statement following the call and **error** will contain the appropriate error code as listed below. If **error** is present and the routine completes successfully, then a zero will be returned for **error.** However if **error** is not specified and the routine encounters an error, the program will terminate with a runtime error.

The following is a list of possible errors that may returned through the optional **error** parameter.

0 = OK
1 = specified file not found
2 = disk is full
3 = end of file encountered
4 = new filename for RENAME already exists
5 = seek error
6 = seek error (but file is closed)
7 = format error in CHAIN or LOAD file

CALL OPEN(unit,'file'{,error})

The OPEN routine is used to open a CP/M file the user may wish to access. **unit** and **file** are required entries. If the CP/M file does not exist and **error** is not specified, then the file will be created. However, if **error** is specified and the file does not exist, the appropriate CP/M error code will be returned and the file will not be opened.

There are 2 special filenames that are recognized by the **OPEN** routine:

CON: used to specify either CP/M console input or output
LST: used to specify CP/M list device

Example

```
CALL OPEN (10,'CON:')
WRITE (10,*) 'A= ',A
```

will output the text to the system console. Files opened with the name CON: can also use a literal in an input statement such as:

```
CALL OPEN (11,'CON:')
READ (11,*) 'INPUT QUANTITY ',QUANT
```

NOTE: The filename (whether a character string or array name) is defined as terminating when:

1) 13 characters are encountered.
2) a NULL is encountered.

CALL LOPEN(unit,'file'{,error})

This subroutine is functionally the same as **OPEN** in that it associates a FORTRAN **unit** with a CP/M file except that the first character of all output records will be processed as carriage control. This is usually used for a listing device such as a printer. The first character of the record will not be output to the file but processed as follows:

first character	action
+	overprint the last record
blank	single skip
0	double skip
-	triple skip
1	page eject

If none of the above characters is present, then single-line spacing will be assumed. Overprinting is implemented by only generating a carriage return at the end of the line (not followed by a line feed). A page eject generates a form feed character (0CH).

The output device that finally prints the output from this file must respond in the following manner:

0DH (carriage return) -- return to beginning of this line
0AH (line feed) -- space 1 line, do not return to beginning of line

0CH (form feed) -- space to the top of the next page

A carriage return must print the line on a line oriented device.

CALL CLOSE(unit)

The CLOSE routine is used as a method of closing FORTRAN files which were previously opened through the OPEN routine. Once the file has been closed, the file number is then available for reuse.

CALL DELET ('file'{,error})

The DELET routine is used by the FORTRAN user to remove a file from the CP/M system. Note that once a file is deleted it cannot be recovered. No error is generated if the file does not exist and **error** is not present.

CALL SEEK (unit,position{,error})

The SEEK routines allow random positioning within a file. The file associated with **unit** will be positioned to **position** which specifies a displacement in bytes from the beginning of the file. If **error** is specified, there are two possible values that may be returned on a seek error. A 6 indicates a seek to a part of the file that doesn't exist, and a 7 indicates a seek to an extent of the file that does not exist. The difference between the two is that if error code 7 is return, the file associated with **unit** is **closed.** The file will have to be re-opened before it can be used again. If **position** is negative, then the file will be positioned to it's end-of-file point. This is normally used for adding to the end of an already existing file. If a file is positioned to end-of-file and then a READ is done, an immediate end-of-file condition will be indicated but a WRITE can be done to extend the file.

CALL RENAME('old file','new file',{error})

The RENAME routine will rename **old file** to **new file.** A runtime error occurs if **old file** does not exist and **error** is not specified or **new file** already exists.

CALL CHAIN('program name'{,error})

The CHAIN routine is used to load in another program overwriting the existing one in memory. This is NOT an overlay, the program that issues the CALL CHAIN will be overwritten by the new program. If **program name** does not exist or the format of **program name** is incorrect and **error** was not specified a **CHAIN FL** runtime error will be produced.

CALL LOAD('file to load',load-type{,error})

The LOAD routine is used to load either a standard CP/M **.HEX** file or a NEVADA ASSEMBLER **.OBJ** file. If **load-type** is zero, then the type of the file to be loaded will be **.HEX**, if **load-type** is non-zero, then the type will be **.OBJ**. This routine can be used to load assembly language routines into memory that can then be accessed through the CALL function. No check is made during the loading process to see whether the object code being read into memory overlays the program or runtime package. It is left up to the programmer to insure that it does not occur. Normally the runtime package occupies memory from 100H to 4000H. If **file to load** does not exist or the format of the file is incorrect and **error** is not specified a **CHAIN FL** runtime error will be produced.

CALL EXIT

The EXIT routine will terminate execution of the FORTRAN program in the same manner as the STOP statement, except that EXIT does not output **STOP** to the system console.

CALL MOVE(count,from,displacement,to,displacement)

The MOVE routine allows direct access to memory for both reads and writes. The **count** specifies the number of bytes to be moved. The arguments **from** and **to** specify either a memory address to be used or a character string to be moved. Which interpretation of **from** and **to** is based on the respective **displacement.** If the **displacement** is negative, then the associated **from** or **to** specifies an address to be used in memory access. If the **displacement** is positive then the **from** or **to** that is associated with it is a string.

Example

```
CALL MOVE(2,A,-1,$CC00,-1)
    This MOVES 2 bytes from the address specified by
    A to address CC00 (HEX).

CALL MOVE(6,'STRING',0,$CC00,-1)
    This MOVES 6 bytes of the string 'STRING' to
    address CC00 (HEX).

CALL MOVE(1024,$CC00,-1,A,0)
    This MOVES 1024 bytes from address CC00 (HEX)
    to the address of A.
```

NOTE: The DOLLAR ($) sign indicates a HEX constant. This HEX constant is converted to floating point notation internally.

CALL DELAY(wait time)

The DELAY routine enables the user to implement a time DELAY of 1/100 of a second to 635.36 seconds. **Wait time** must be in range of 0 to 65535 with 0 being the maximum delay time, 1 being the shortest and 65535 being 1/100 less than 0. This time is based on a 2 MHZ 8080 processor.

CALL CIN(char)

The CIN routine enables the user to obtain a single character from the system console. The character is returned as the leftmost byte of **char** in 8 bit binary format. The left most bit of value read will be zeroed.

Example

```
C WAIT FOR A CARRIAGE RETURN (ODH) FROM THE CONSOLE
C BEFORE CONTINUING.
  80  CALL CIN(CHAR)
      IF (COMP(CHAR,#0D00,1) .NE. 0)GO TO 80
```

CALL CTEST(status)

The CTEST routine is used to test the status of the system console. A zero is returned in **status** if there is no character ready to input on the system console. A one is returned if there is a character.

Example

```
C WAIT IN A LOOP UNTIL A CHARACTER IS HIT ON THE
C SYSTEM CONSOLE, THEN CHECK THE CHARACTER FOR A
C LINE FEED (0AH) BEFORE CONTINUING.
    ARAND=.3478
 10 ARAND=RAND(ARAND)
    CALL CTEST(STATUS)
    IF (STATUS .EQ. 0)GO TO 10
C
C CHARACTER HIT, READ IT
C
    CALL CIN(CHAR)
    IF (COMP(CHAR,#0A00,1) .NE. 0)GO TO 10
```

CALL OUT (port,value)

This routine allows access to the 8080 output ports. **Value** will be converted to an 8 bit number and output to **port.**

Example

```
CALL OUT(10,1)
```

CALL SETIO(new I/O)

This routine allows changing how the runtime package performs console I/O. The default method is setup using the **CONFIG** program, however it can be changed as follows:

new I/O = 0 to use direct BIOS I/O
new I/O = 2 to use CP/M function 1&2
new I/O <> 0 or 2 to use CP/M function 6 (2.0 only).

A=INP(port)

This routine allows access to the 8080 input ports. This routine is a function whose value is that which is read from the port specified as **port.**

Example

```
I=INP(10)
```

A=CALL(address,argument)

The CALL function causes execution of assembly language routines that have been loaded into memory (usually by the LOAD subroutine). **Address** is the memory location to be call'ed. **argument** will be converted to a 16 bit binary number and then passed to the called routine in both the BC and DE register pairs. The assembly routine places the value to be returned in register pair HL. The return address is placed on the 8080 stack and the call'ed routine can just issue a standard RET instruction to return to the FORTRAN program.

A=CBTOF(from,displacement{,8-bit})

The CBTOF function is used to convert either a 16 bit or 8 bit binary number to its equivalent floating point value. The number to be converted is located at **from+displacement** if **displacement** is positive. If **displacement** is negative, then **from** contains the address to be used. The number is assumed to be 16 bit value (store in standard 8080 format) unless **8-bit** is present, in which case it will be assumed to be an 8 bit value. The binary number is considered to be unsigned.

Example

```
BIOS=CBTOF($0006,0)-3
```

gets the base address of the CP/M bios.

11.0.0 Appendix

11.1.0 Statement Summary

variable = expression

Assigns the value of the **expression** to the **variable.**

ACCEPT input list
Reads values from the system console and assigns them to the variables in the **input list.**

ASSIGN n TO V
Assigns a statement label to a variable to be used in an assigned go to.

BACKSPACE unit
Positions the specified unit to the beginning of the previous record.

BLOCK DATA
Begin a BLOCK DATA subprogram for initializing variables in COMMON.

CALL name(argument list)
Call the subroutine passing the argument list.

COMMON /label1/list1 /label2/list2
Declares the variables and array that are to be placed in COMMON amongst the various routines.

CONTINUE
Causes no action to take place, usually used as the object of a GOTO or DO loop.

COPY filename
The specified filename is inserted into the source at the point of the COPY statement.

CTRL DISABLE
Disables program termination by control/c from the console.

CTRL ENABLE
Enables program termination by control/C from the console. Control C being enabled is the default.

DATA /var1/const1,const2/var2/c1,c2,.../
Initializes the specified variable, array element or arrays to the specified constants.

DIMENSION v(n1,n2,..),v2(n1,n2,..)
Sets aside space for arrays **v** and **v2.**

DO n i=n1,n2,n3

Executes statements from DO to statement **n**, using **i** as index, increasing or decreasing from **n1** to **n2** by steps of **n3**.

DUMP /id/ output list

When a runtime error occurs, displayed **id** and items in **output list.**

END

This statement must be the last statement of every routine.

ENDFILE unit

Write an end of file at the current position of **unit.**

ERRCLR

Clears the effect of the ERRSET statement.

ERRSET n,v

When a runtime error occurs, control goes to the statement labelled **n** with variable **v** containing the error code.

FORMAT (field specifications)

Used to specify input and output record formats.

FUNCTION name(argument list)

Begins the definition of a function subprogram.

GO TO n

Transfer control to the statement labelled **n.**

GO TO (n1,n2,...),v

The COMPUTED GOTO transfers control to **n1** if **v=1,n2** if **v=2,** etc.

GO TO v,(n1,n2,..)

The ASSIGNED GOTO transfers control to statement **n1, n2,..** depending on the value of **v. V** must have appeared in an ASSIGN statement.

IF (e)n1,n2,n3

Transfer control to **n1** if **e<0, n2** if **e=0** or **n3** if **e>0.**

IF (e)statement

Executes **statement** if the value of expression **e** is true (non-zero).

IF (e) THEN statement1 ELSE statement2 ENDIF

Executes blocks of statements **statement1** if **e** is true, or block of statements **statement2** if **e** is false.

IMPLICIT type(letter list)

Changes the default type of variable that start with the letters in the **letter list.**

INTEGER v1,v2,..
Declares **v1, v2,** etc, to be integer variables.

LOGICAL v1,v2,...
v1, v2, etc, to be logical variables.

PAUSE 'character string'
Suspends program execution until any key is hit, displaying PAUSE and **character string.**

READ (unit,format{,ERR=}{,END=}) input list
Reads values from **unit** according to **format** and assigns them to the variables in **input list.**

REAL v1,v2,..
Declares **v1, v2,** etc, to be real variables.

RETURN
Returns control from a subprogram to the statement following either the call or the function reference.

RETURN i
Return control from a subprogram to statement **i** in the calling routine.

REWIND unit
The file associated with **unit** is closed, then reopened at the beginning of the same file.

STOP 'character string'
Terminates program execution and displays **character string** on the system console.

STOP n
Terminates program execution and displays **n** on the system console.

SUBROUTINE name(argument list)
Begins the definition of a subroutine subprogram.

TRACE OFF
Turns statement tracing off.

TRACE ON
Turns statement tracing on.

TYPE output list
Displays the value of the variables in **output list** on the system console.

WRITE (unit,format{,ERR=}) output list
Writes the values of the variable in **output list** to **unit** according to **format.**

11.2.0 NON-STANDARD FUNCTIONS

A=COMP(string1,string2,length)

The COMP routine is used to compare character strings in the following manner.

A=COMP('string1','string2',length).

The strings will be compared on a byte basis for a byte count of length. The routine returns the following:

-1 if **string1** < **string2**
 0 if **string1** = **string2**
+1 if **string1** > **string2**

A=CALL(address,value)

The CALL routine allows assembly language programs to be CALLed and to be passed an argument. **Value** will be converted to a 16 bit binary number and passed to the assembly routine (at **address**) in both REGS BC and DE. The value of the function is passed back in REG HL. The return address is on the top of the 8080 stack.

CALL BIT(variable,bit displacement,'S')
'R'
'F'
'T',value

The BIT subroutine allows the setting (**S**), resetting (**R**), flipping (**F**) or testing (**T**) of individual bits.

The bit at **bit displacement** from the start of **variable** will be set if **S** is specified, reset if **R** is specified, flipped (1 will become 0 and 0 will become 1) if **F** is specified and finally the value of the selected bit will be returned in **value** if **T** is specified. **Value** must be present only for **T**. Displacement is specified starting with the leftmost bit.

11.3.0 Available Functions

Name	Function	Arg	result	argument
SIN	Sine(x)	1	real	real
COS	Cosine(x)	1	real	real
TAN	Tangent(x)	1	real	real
ATAN	Arctangent(x)	1	real	real
ATAN2	Arctangent(y/x)	2	real	real
ALOG	Log base e (x)	1	real	real
ALOG10	Log base 10 (x)	1	real	real
MOD	Remainder (x/y)	2	integer	integer
AMOD	Remainder (x/y)	2	real	real
SQRT	Square Root (x)	1	real	real
FLOAT	Make real (x)	1	real	integer
IFIX	Truncate (x)	1	integer	real
ABS	Absolute (x)	1	real	real
IABS	Absolute (x)	1	integer	integer
RAND	Random Number (x)	1	real	real 0.0<R<1.0
SGN	Sign of (x)	1	-1,0,+1	real/integer
EXP	e**(x)	1	real	real
COMP	Compare strings	3	either	real
CALL	CALL assembly pgm	2	either	real
AMAX0	Maximum	?	either	either
AMAX1	Maximum	<255	either	either
MAX0	Maximum	<255	either	either
MAX1	Maximum	<255	either	either
AMIN0	Minimum	<255	either	either
AMAX0	Minimum	<255	either	either
MAX0	Minimum	<255	either	either
MAX1	Minimum	<255	either	either
BIT	Bit handling	3/4	either	either

Most of the above functions as ANSI standard except for RAND. This function behaves as if it were returning an entry from a table of random numbers. The argument of RAND determines which entry of this table will be returned:

Rand Arg.	Value returned for RAND
0	The next entry in the table
-1	The first entry in the table. Also the pointer for the next entry (arg=0) is reset to the second entry in the table.
n	Returns the table entry following n.

11.4.0 Summary of System subroutines

CALL BIT (variable,disp,code)
Set, resets or flips bit 0+**disp** of **variable** according the **code.**

CALL CBTOF(locl,displ,loc2{,flag})
Converts a binary number to its floating point equivalent.

CALL CHAIN('program name'{,error})
Loads another program and executes it.

CALL CIN(var)
Reads a single character from the system console.

CALL CLOSE(unit)
Close the file associated with **unit.**

CALL CTEST(status)
Determines if a character has been entered on the system console.

CALL DELAY(time)
Delays execution the specified time/100 seconds.

CALL DELET('filename'{,error})
Delete the specified **filename** from the disk.

CALL EXIT
Terminates program execution.

CALL MOVE(n,locl,displ,loc2,disp2)
Moves **n** bytes from **locl** to **loc2.**

CALL OPEN(unit,'filename'{,error})
Opens the specified **filename** and associates it with **unit.**

CALL LOAD('filename',load-type{,error})
This routine is used to load a file of type .HEX or .OBJ into memory depending on the value of **load-type.**It is usually used to load assembly language routines into memory. No check is made to see if the code that is loaded into memory would override the program or CP/M.

CALL LOPEN(unit,'filename'{,error})

Opens the specified **filename** and associates it with **unit.** This file is also treated as a printer file with the first character of each output record controlling paper movement.

CALL OUT(port,value)

value is converted to an 8 bit number and output to port **port.**

CALL RENAM('old name','new name'{,error})

Renames **old name** to be **new name.**

CALL SETIO (new I/O)

Allows changing the way that console I/O is performed during program execution.

CALL SEEK (unit,position)

Positions the file associated with **unit** to the byte position specified by **position.** Positions the file to it's end if **position** is negative.

11.5.0 RUNTIME ERRORS

During execution of a program, there are numerous conditions that can occur which cause program termination. When one of these conditions is encountered, a RUNTIME ERROR message will be generated to the system console file. The message has the format:

Runtime error: XXXXXXXX, called from loc. YYYYH

Pgm was executing line LLLL in routine NNNN

where: **XXXXXXXX** is the ERROR, **YYYY** is the memory location of the CALL to the runtime package in which the error occurred.

The second line of the error message will be generated as a traceback of CALL statements that have been executed. The **LLLL** is the FORTRAN generated line number (shown on the listing of the source from the compiler) of the statement which caused the error, and **NNNN** if the name of the routine in which that line number corresponds. The line number will be output as **????** if the **X** option was not specified on the $OPTIONS statement for a given routine. If multiple 'PGM WAS ...' lines are printed, the first one specifies the line in which the error actually occurred.

Runtime Errors

INT RANG

INTEGER OVERFLOW: result greater than 8 digits

CONVERT

16 BIT CONVERSION ERROR: in converting a number from integer to internal 16 bit binary, an overflow has occurred. This can occur on all statements associated with I/O (unit number), subscript evaluation and anywhere that a number has to be converted from floating to 16 BIT binary.

ARG CNT

ARGUMENT COUNT ERROR: a subprogram call had too many or too few arguments.

COM GOTO

COMPUTED GO TO INDEX OUT OF RANGE: the variable specified in a computed GOTO is either zero or greater than the number of statement labels specified.

OVERFLOW

FLOATING POINT OVERFLOW: the result of a floating point operation has resulted in a number whose value is too large to be stored.

DIV ZERO

DIVIDE BY ZERO: an attempt has been made to divide by zero.

SQRT NEG

SQRT OF NEGATIVE NUMBER: argument of the square root function is negative.

LOG NEG

LOG OF NEGATIVE NUMBER: argument of the log either (ALOG or ALOG10) function is negative.

CALL PSH

CALL STACK PUSH ERROR: this error is caused by a recursive subprogram CALLS of depth greater than 36. Only in very special cases should a subprogram CALL itself or one of those that has CALLED it.

CALL POP

CALL STACK POP ERROR: this error should never occur (This means that a RETURN has been executed that does not have a corresponding CALL or FUNCTION reference. Usually caused by user assembly language programs).

CHAIN FL

CHAIN FILE ERROR: the filename specified in a call to the CHAIN or LOAD routine was not found on the disk.

ILL UNIT

ILLEGAL UNIT NUMBER (<2 OR >15): an illegal unit number has been passed to one of the I/O routines.

UNIT OPN

UNIT ALREADY OPEN: this is generated by the OPEN routine when an attempt to open a file on an already open FORTRAN logical unit.

DSK FULL

DISK FULL: either the disk is full or the directory is full.

UNIT CLO

UNIT CLOSED: a reference has been made to a FORTRAN unit number that is not OPEN.

CON BIN

BINARY I/O TO CONSOLE: binary I/O is not supportted to the system console.

LINE LEN

LINE LENGTH ERROR: an attempt has been made to READ or WRITE a record whose length exceeds 250 characters. This count also includes a carriage return at the end of the line.

FORMAT

FORMAT ERROR: an unrecognized or invalid FORMAT specification has been encountered in a FORMATTED READ or WRITE.

I/O ERR

I/O ERROR: an error occurred during a READ or WRITE operation and the ERROR label was not specified in the statement. It will also be generated during a READ if END OF FILE is encountered and an EOF label was not specified.

ILL CHAR

ILLEGAL CHARACTER: an illegal character has been encountered during a READ.

I/O LIST

INVALID I/O LIST: this error indicates an error in the I/O list specification of a formatted WRITE or READ. This error will only occur if a user assembly program does not construct the I/O list correctly. It will never occur from FORTRAN generated code.

ASN GOTO

ASSIGNED GO TO ERROR: the value of the variable specified in an ASSIGNED GO TO does not match that of one of the statement labels listed.

CONTRL/C

CONTROL/C error: CONTROL/C was hit and the CONTROL/C was not trapped.

INPT ERR

INPUT ERROR: during a READ an invalid character has been encountered for the number being processed. This will be generated for such things as: two decimal points in a number, an E in an F type field, decimal point in an I type field, etc.

FILE OPR

FILE OPERATION ERROR: an error has occurred while trying to do some file operation, such as renaming when the new file already exists.

SEEK ERR

SEEK ERROR: an error has occurred while positioning a file to the specified position and no error variable was specified in the CALL.

11.6.0 COMPILE TIME ERRORS

The following is a list of errors that may occur during the compilation of a FORTRAN program. If the **G** option is not selected a two digit error number will be printed instead. This number can be found at the beginning of each line.

00 *FATAL* compiler error
01 Syntax error, 2 operators in a row
02 unexpected continuation (column 6 not blank or 0)
03 input buffer overflow (increase B= compiler option)
04 invalid character for FORTRAN statement
05 unmatched parenthesis
06 statement label > 99999
07 invalid character encountered in statement label
08 invalid HEX digit encountered in constant
09 expected constant or variable not found
0A 8 bit overflow in constant
0B unidentifable statement
0C statement not implemented
0D quote missing
0E SUBROUTINE/FUNCTION/BLOCK DATA not first statement in routine
0F columns 1-5 of continuation statement are not blank
10 cannot initialize BLANK COMMON
11 RETURN is not valid in main program
12 syntax error on unit specification
13 missing comma after) in COMPUTED GO TO
14 missing variable in COMPUTED GO TO
15 invalid variable in ASSIGNED/COMPUTED GO TO
16 invalid LITERAL, no beginning quote
17 number of subscripts exceeds maximum of 7
18 invalid SUBROUTINE or FUNCTION name
19 subscript not POSITIVE INTEGER CONSTANT
1A FUNCTION requires at least one argument
1B syntax error
1C invalid argument in SUBROUTINE/FUNCTION call
1D first character of variable not alphabetic
1E ASSIGNED/COMPUTED GOTO variable not integer
1F label has already defined
20 specification of array must be integer
21 invalid variable name
22 invalid DIMENSION specification
23 dimension specification is invalid
24 variable has already appeared in type statement
25 invalid subroutine name in CALL
26 SUBPROGRAM argument cannot be initialized
27 improperly nested DO loops
28 unit not integer constant or variable
29 Array size exceeds 32K
2A invalid use of unary operator
2B variable DIMENSION not valid in MAIN program
2C variable dimensioned array must be argument
2D DO/END/LOGICAL IF cannot follow LOGICAL IF

2E undefined label
2F unreferenced label
30 FUNCTION or ARRAY missing left parenthesis
31 invalid argument of FUNCTION or ARRAY
32 DIMENSION specification must precede first executable statement
33 unexpected character in expression
34 unrecognized logical opcode
35 argument count for FUNCTION or ARRAY wrong
36 *COMPILER ERROR* poped off bottom of operand stack
37 expecting end of statement, not found
38 statement too complex; increase P and/or O table
39 invalid delimiter in ARITHMETIC IF
3A invalid statement number in IF
3B HEX constant > FFFF (HEX)
3C replacement not allowed within IF
3D multiple assignment statement not implemented
3E subscripted-subscripts not allowed
3F subscript stack overflow; increase P= or O=
40 missing left (in READ/WRITE
41 invalid unit specified
42 invalid FORMAT, END= or ERR= label
43 invalid element in I/O list
44 built-in function invalid in I/O list
45 cannot subscript a constant
46 variable not dimensioned
47 invalid subscript
48 missing comma
49 index in IMPLIED DO must be a variable
4A invalid starting value for IMPLIED DO
4B invalid ending value of IMPLIED DO
4C invalid increment of IMPLIED DO
4D illegal use of built-in function
4E variable cannot be dimensioned in this context
4F invalid or multiple END= or ERR=
50 invalid constant
51 exponent overflow in constant
52 invalid exponent
53 character after . invalid
54 integer overflow
55 integer underflow (too small)
56 missing = in DO
57 string constant not allowed
58 invalid variable in DATA list
59 DATA symbol not used in program, line
5A invalid constant in DATA list
5B error in DATA list specification
5C FUNCTION invalid in DATA list
5D no filename specified on COPY
5E runtime format not array name
5F DUMP label invalid or more than 10 characters
60 more than 1 IMPLICIT is not allowed
61 IMPLICIT not first statement in MAIN, 2nd statement in SUBPROGRAM
62 data type not REAL, INTEGER or LOGICAL

63 illegal IMPLICIT specification
64 improper character sequence in IMPLICIT
65 variable already DIMENSIONED
66 Q option must be specified for ERRSET/ERRCLR
67 Hex constant of zero (0) invalid in I/O stmnt
68 Argument cannot also be in COMMON
69 Illegal COMMON block name
6A Variable already in COMMON
6B Array specification must precede COMMON
6C Executable statement invalid in BLOCK DATA
6D Hex constant of 27H (') invalid in FORMAT
6E Invalid number following STOP or PAUSE
6F invalid TRACE statement (operand not ON/OFF)
70 invalid IOSTAT= variable
71 missing , in ENCODE/DECODE
72 invalid label in ASSIGNED GOTO
73 invalid variable in ASSIGNED GOTO
75 label not allowed on this statement
75 multiple RETURN not valid in FUNCTION
76 UNUSED
77 no matching IF-THEN for ELSE or ENDIF
78 invalid ELSE or ENDIF
79 missing ENDIF
7A initialization of non-COMMON variable
7B UNUSED
7C UNUSED
7D UNUSED
7E UNUSED
7F UNUSED

80 *FATAL* no program to compile
81 *FATAL* missing $OPTIONS statement
82 *FATAL* missing = in $OPTIONS statement
83 *FATAL* invalid digit in number in $OPTIONS
84 *FATAL* value exceeds 255 in $OPTIONS
85 *FATAL* COMMON table overflow, increase C=
86 *FATAL* unknown option (letter before =)
87 *FATAL* missing END statement
88 *FATAL* LABEL TABLE overflow, increase L=
89 *FATAL* SYMBOL TABLE overflow, increase S=
8A *FATAL* ARRAY STACK overflow, increase A=
8B *FATAL* DO LOOP STACK overflow, increase D=
8C *FATAL* stack overflow (compiler error)
8D *FATAL* stack overflow (compiler error)
8E *FATAL* internal tables exceed user memory
8F *FATAL* MEMORY ERROR
90 *FATAL* OPEN error on COPY file
91 *FATAL* too many routines to compile (> 62)
92 *FATAL* no more room to store DATA statements
93 *FATAL* IF-THEN stack overflow, increase I=
94 *FATAL* Nested "COPY" statements not permitted
95 *FATAL* Disk write error (disk probably full)
96 *FATAL* Cannot close file (disk probably full)
97 *FATAL* Input file not found
98 *FATAL* Invalid drive specifier
99 *FATAL* No filename found on COPY statement
9A *FATAL* File specified on COPY not found

11.7.0 ASSEMBLY LANGUAGE INTERFACE

ASSEMBLY statements can be directly inserted into a FORTRAN program by preceding the statement with an ASTERISK (*). The line that contains that asterisk will be directly output to the assembly file without further processing (the asterisk is deleted first). Because of the nature of the FORTRAN compiler (it actually reads one statement ahead of where it is processing) it is ALWAYS a good idea to put a **CONTINUE** statement immediately preceding the first assembly statement in each separated group of assembly statements. The **CONTINUE** will cause the assembly statements to be inserted at the expected place. FORTRAN maintains nothing in the registers between statements, but does use the 8080 stack for saving RETURN addresses for user called FUNCTIONS and SUBROUTINES.

11.8.0 GENERAL COMMENTS

1. In the description of the individual routines, anywhere that a character string is specified, a variable or array name can be used. The variable or array can be set to the desired character string.

2. A variable can be set to a character string using an assignment statement such as:

```
A='STRING'
```

No more than 6 characters will be retained for any variable and if less than 6 will be zeroed filled in the low order bytes of the variable.

3. If a variable or array name is used to reference a CP/M file (such as in the OPEN routine) the filename itself within the variable (or array) is terminated after:

1) first 13 characters,
2) a NULL is encountered.

4. HEX constants can be used anywhere that a constant or variable is permitted. A hex constant is specified by preceding it by a dollar sign ($). Examples follow:

```
A=$E060
A=-$CC00
```

Hex constants are limited to a maximum value of FFFF. An error is generated is a hex constant exceeds this limit. Internally a hex constant is treated as any other INTEGER constant would be.

5. A HEX constant that is preceded by a # instead of a $ will be stored internally in binary format in the first two bytes of the variable. Numbers of this form should be put through the FLOATING MATH package. The number is stored in standard 8080 format (HIGH byte followed by LOW byte).

6. A backslash (\) can be used in a literal to specify an 8 bit binary constant to be inserted at that point. The constant is enclosed in back slashes and is assumed to be a HEX constant. For example:

```
      A='THIS \32\ IS AN EXAMPLE'

      CALL OUTIT(3,1,'\7F\\FF\',2,32)

   10 FORMAT ('IT IS ALLOWED \1\ HERE \FF\ ALSO')
```

NOTE The backslash is the default character and can be changed using the CONFIG program.

7. 16 Fortran files may be open at any one time (file numbers 0-15). Remember that files 0 and 1 are permanently open.

11.9.0 USE OF THE NORTH STAR FLOATING POINT BOARD

A feature of NEVADA FORTRAN is that it will directly use the North Star floating point board if one is installed in the system that the program is executed on. Use the CONFIG program to specify the 2 memory addresses that the floating point uses. If you should specify that you have a floating point board, at runtime the runtime package checks to see if the floating point board is actually in the system and if not, then uses the software floating point routines instead of the hardware. One disadvantage of using the floating point board is that the range of real number decreases to 10**-63 to 10**+63. Any number generated outside this range will produce an overflow error message.

11.10.0 COMPARISON OF NEVADA FORTRAN AND ANSI FORTRAN

NEVADA FORTRAN includes the following extensions to version X3.9-1966 of ANSI Standard FORTRAN:

1. Free-format input and output
2. IMPLICIT statement for setting default variable types
3. Options end-of-file and error branches in READ and WRITE statements.
4. COPY statement to insert source files into a FORTRAN program.
5. Direct inline assembly language.
6. Access to file system for such functions as creating, deleting and renaming files.
7. Random access on a byte level to files.
8. Access to absolute memory locations.
9. Program controlled time delay.
10. A pseudo random number generator function.
11. Program control of runtime error trapping.
12. Ability to chain a series of programs.
13. Ability to load object code into memory.
14. CALL function to execute previously loaded code.
15. Program tracing.
16. IF-THEN-ELSE statement
17. Enabling and disabling console abort of program.
18. ENCODE and DECODE memory to memory I/O.
19. Multiple returns from subroutines.
20. K format specification.

NEVADA FORTRAN does not included the following features of ANSI standard FORTRAN:

1. Double Precision, double precision functions, statements, and format specifications.
2. Complex numbers, complex statements and functions.
3. EQUIVALENCE statement.
4. Extended DATA statement of the form:

```
DATA A,B,C/1,2/3/
```

5. The D and P format specifications.
6. Statement functions.
7. Only the first five characters of function or subroutine names or COMMON block labels are retained. In other words, only the first 5 characters of the name is retained.
8. The following are reserved names for functions, subroutines or COMMON block names:

```
A, B, C, D, E, H, L, M, SP, PSW
```

11.11.0 SAMPLE PROGRAMS

On the following pages you will find listings of the sample programs that may have been included on your diskette.

```
C
C "CHAIN.FOR"
C
C THIS ROUTINE DEMONSTRATED THE 'CHAIN' FUNCTION, ALL IT
C DOES IS REQUEST THE NAME OF THE PROGRAM TO CHAIN TO
C AND THEN CHAIN.
C
        DIMENSION IF(3)
        TYPE 'FILE?'
C
C GET THE FILENAME TO CHAIN TO
C
        READ (0,1) IF
1       FORMAT (3A6)
C
C CHAIN TO IT
C
        CALL CHAIN (IF,IER)
C
C ONLY GETS HERE IF AN ERROR HAPPENS
C
        TYPE 'ERROR FROM CHAIN = ',IER
        CALL EXIT
        END
```

```
OPTIONS X
C
C "DUMP.FOR"
C
C THIS PROGRAM DEMONSTRATED THE USE OF THE DUMP STATEMENT.
C
C CALL 'X' FOR TRACEBACK PRINTOUT (JUST FOR SHOW)
C
        CALL X
        END
OPTIONS X
        SUBROUTINE X
C
C DEFINE THE DUMP STATEMENT TO BE USED IN CASE OF AN
C ERROR, WITH DUMP ID OF 'ROUTINE-X'
C

        DUMP /ROUTINE-X/ I,J,K
        I=1
        J=2
        K=I+J
C
C CREATE AN ERROR TO CAUSE DUMP STATEMENT TO BE ACTIVE
C
        Z=1/0
        END
```

```
OPTIONS X
C
C "GRAPH.FOR"
C
C GRAPH SINE FUNCTION FROM -PI TO PI IN INCREMENT OF .12
C
        DIMENSION LINE(70)
        INTEGER WHERE
C
C OPEN UNIT 6 TO WRITE TO CONSOLE
C
        CALL OPEN (6,'CON:')
C
C WRITE TITLE
C
        WRITE (6,2)
2       FORMAT (28X,'GRAPH OF SIN')
        TYPE
        TYPE
C
C SET PI AND -PI
C
        PI=3.1415926
        MPI=-PI
C
C MAIN LOOP
C
        DO 100 ANGLE=MPI,PI,.12
C
C FIGURE OUT WHICH ELEMENT IN ARRAY SHOULD BE SET TO *,
C SIN RETURNS -1 TO 1 WHICH IS CONVERTED TO -35 TO 35
C AND THEN OFFSET SO FINAL RANGE IS 1 TO 70
C
        WHERE=SIN(ANGLE)*35+35
C
C FIGURE OUT HOW MUCH TO BLANK IN THE OUTPUT ARRAY
C
        IBLANK=MAX0(35,WHERE)
C
C AND BLANK IT
C
        DO 15 I=1,IBLANK
15      LINE(I)=' '
C
C HMM... WHICH SIDE OF ZERO ARE WE ON?
C
        IF (WHERE .GT. 35)  THEN
C
C RIGHT SIDE
C
                        DO 20 I=36,WHERE
20                      LINE(I)='*'
                            ELSE
```

```
C
C LEFT SIDE
C
                        DO 30 I=WHERE,35
30                      LINE(I)='*'
                            ENDIF
C
C SET "ZERO"
C
        LINE(35)='+'
C
C AND THE SIN VALUE
C
        LINE(WHERE)='*'
C
C IF THIS VALUE IS < 35, SET SO WE OUTPUT TO ZERO LINE
C
        IF (WHERE .LE. 35)WHERE=35
C
C AND FINALLY OUTPUT THE LINE
C
        WRITE (6,21) (LINE(I),I=1,WHERE)
21      FORMAT (70A1)
100     CONTINUE
        CALL EXIT
        END
```

```
C
C "LOAD.FOR"
C
C THIS ROUTINE DEMONSTRATED THE USE OF THE 'LOAD' ROUTINE
C TO LOAD AN ASSEMBLY LANGUAGE FILE INTO MEMORY AND
C THEN CALL IT FORM FORTRAN
C
       INTEGER A
C
C FIND OUT WHICH ONE TO LOAD
C
       TYPE 'Enter 0 to "LOAD" LD.HEX'
       TYPE 'Enter 1 to "LOAD" LD.OBJ'
       ACCEPT 'Which one: ',LTYPE
C
       IF (LTYPE .EQ. 0) THEN
                         TYPE '"LOAD"ing "LD.HEX"'
                          ELSE
                         TYPE '"LOAD"ing "LD.OBJ"'
                          ENDIF
C
C MUST LOAD "LD.HEX" OR "LD.OBJ" INTO MEMORY
C BEFORE WE CAN CALL IT
C
       CALL LOAD ('LD',LTYPE,IER)
C
       TYPE 'ERROR FOR LOAD=',IER
C
C CHECK THE RETURNED ERROR CODE FROM LOAD
C
       IF (IER .NE. 0)STOP 'LOAD ERROR'
C
C "CALL" THE ROUTINE
C
       A=CALL ($8000,1)
C
C RESULT SHOULD BE 2
C
       TYPE 'THE RESULT OF THE ASSEMBLY ROUTINE IS: ',A
       CALL EXIT
       END
```

```
;
; "LD.ASM"
;
; THIS ROUTINE IS USED BY "LOAD.FOR", ALL IT DOES IS TO
; DOUBLE THE NUMBER SENT TO IT
; NUMBER IS PASSED IN DE FROM FORTRAN PROGRAM AND RESULT
; IS PASSED BACK IN HL TO FORTRAN PROGRAM
;
       ORG     8000H
       PUSH    D       ;NUMBER FROM FORTRAN PROGRAM
       POP     H       ;GET IT TO HL
       DAD     H       ;HL*2
       RET             ;RETURN IT
```

```
OPTIONS X,Q
C
C "RAND.FOR"
C
C THIS PROGRAM GENERATES A SEQUENCE OF RANDOM NUMBERS,
C DIVIDES THEM INTO 10 INTERVALS AND COUNTS HOW MANY
C RANDOM NUMBER FALL INTO EACH INTERVAL.  FINALLY IT
C PRINTS OUT THE COUNTS OF EACH INTERVAL.
C
        DIMENSION NUM(10)
        DATA NUM/10*0/
        INTEGER T,A,D,FLAG,TIME(6),DATE(6),START,END
99      DO 50 I=1,10
50      NUM(I)=0
        ACCEPT 'How many? ',K
        DO 1 I=1,K
        L=RAND(0)*10+1
1       NUM(L)=NUM(L)+1
        TYPE NUM
        GO TO 99
        END
```

```
OPTIONS X,Q
C
C "SEEK.FOR"
C
C THIS PROGRAM DEMONSTRATES RANDOM ACCESS I/O
C
C IT FIRST WRITES A FILE OF NUMBERS, THEN REQUESTS
C A RECORD, READ NUMBER THAT THE RECORD CONTAINS,
C ADDS 1 TO THE NUMBER READ AND WRITES IT BACK INTO
C THE SAME RECORD
C
        ERRSET 500,I
        CALL DELET('TEST')
C
C OPEN THE TEST FILE
C
        CALL OPEN (2,'TEST')
C
C READ HOW MANY RECORDS TO CREATE
C
        ACCEPT 'HOW MANY RECORDS? ',K
C
C WRITE THE FILE
C
        DO 1 I=0,K
1       WRITE (2,2) I
2       FORMAT (I5)
        TYPE 'FILE WRITTEN'
        TYPE
        REWIND 2
        GO TO 10
        CALL OPEN(2,'TEST')
C
C REQUEST RECORD TO DISPLAY
C
10      ACCEPT 'WHICH RECORD? ',K
C
C POSITION THE FILE (EACH RECORD IS 7 CHARACTERS,
C 5 FOR NUMBER, 1 FOR CARRIAGE RETURN AND 1 FOR LINE FEED
C
        CALL SEEK (2,7*K,IER)
C
C CHECK THE ERROR CODE
C
        IF (IER .NE. 0) THEN
                        TYPE 'SEEK ERROR, CODE= ',IER
                        CALL CLOSE (2)
                        CALL DELET ('TEST')
                        STOP
                        ENDIF
```

```
C
C READ THE CURRENT VALUE
C
        READ (2,2) I
        TYPE 'CURRENT VALUE OF RECORD ',K,' IS ',I
C
C POSITION BACK TO THE SAME RECORD
C
        CALL SEEK(2,7*K)
        I=I+1
C
C WRITE THE UPDATED VALUE
C
        WRITE (2,2) I
        GO TO 10
C
C TRAP ERROR
C
500     TYPE '*** ERROR TRAPPED ***'
        TYPE 'ERROR CODE = ',I
        CALL CLOSE (2)
        CALL DELET ('TEST')
        STOP 'ERROR EXIT'
        END
```

```
C
C "SORT.FOR"
C
C THIS ROUTINE IS A DEMONSTRATION OF A SHELL SORT
C
        INTEGER T,A,D,FLAG,TIME(6),DATE(6),START,END
        DIMENSION A(2000)
        TYPE 'Shell sort'
        TYPE
C
C GET HOW MANY NUMBERS TO SORT
C
88      ACCEPT 'How many numbers (2-2000) ',NN
        IF (NN .LT. 2.OR.NN .GT. 2000)STOP
C
C GENERATE ARRAY OF NUMBERS TO SORT
C
        DO 10 I=1,NN
10      A(I)=(RAND(0)*NN)+1
        TYPE 'Starting sort'
        D=NN
        FLAG=0
C
100     D=IFIX((D+1)/2)
C
C TYPE OUT INTERMEDIATE STUFF
C
        TYPE 'D=',D
C
110     ND=NN-D
        DO 150 N=1,ND
        IF (A(N) .LE. A(N+D))GO TO 150
        NPD=N+D
        T=A(N)
        A(N)=A(NPD)
        A(NPD)=T
        FLAG=1
C
150     CONTINUE
        IF (FLAG .EQ. 1)THEN
                        FLAG=0
                        GO TO 110
                        ENDIF
        IF (D .GT. 1)GO TO 100
        TYPE 'All done'
        TYPE
C
C TYPE OUT SORTTED ARRAY
C
        TYPE (A(I),I=1,NN)
        GO TO 88
        END
```

```
OPTIONS X,Q
C
C "TRACE.FOR"
C
C THIS ROUTINE DEMONSTRATES THE USE OF THE 'TRACE' AND
C 'ERROR' TRAPPING FUNCTIONS
C
        TYPE 'STARTING EXECUTION'
C
C SET ERROR TRAPPING: ON ERROR GO TO STATEMENT 500 WITH
C ERROR CODE IN VARIABLE I
C
        ERRSET 500,I
10      CONTINUE
C
C TURN TRACING OFF
C
        TRACE OFF
C
C GET AN INPUT # FROM THE USER
C
        ACCEPT '#: ',K
C
C IF <0, TERMINATE
C
        IF (K .LE. 0)GO TO 99
C
C IF INPUT # > 100, THEN TURN TRACING ON
C
        IF (K .GT. 100)TRACE ON
C
C AND OUTPUT THE NUMBERS, TO SEE EFFECT OF THE
C ERROR TRAPPING, HIT CONTROL-C
C
        DO 20 I=1,K
20      TYPE I
        GO TO 10
C
99      TYPE 'DONE'
        STOP
C
C ERROR TRAPPING HAPPENS HERE
C
500     TYPE 'ERROR TRAPPED, IER= ',I
        END
```

11.12.0 SAMPLE PROGRAM COMPILATIONS AND EXECUTION

On the following pages you will find examples of the compilation and execution of several of the sample programs listed above. The following notes refer to the next few pages:

1) input is underlined.

2) CP/M output is printed in **bold**

3) FORTRAN output (either compiler or execution) is neither underlined or bold.

4) notes are in {}.

```
B>FORT GRAPH.XBB   {compile with listing to console,
                    .ASM and .OBJ to drive B}


NEVADA FORTRAN 2.00 (MOD 0)
Copyright (C) 1979, 1980, 1981, 1982 Ian Kettleborough

***** NEVADA Fortran 2.00 (Mod 0) ** Compiling File: GRAPH.FOR *****

0001 OPTIONS X
     C
     C GRAPH SINE FUNCTION FROM -PI TO PI IN INCREMENT OF .12
     C
0002          DIMENSION LINE(70)
0003          INTEGER WHERE
     C
     C OPEN UNIT 6 TO WRITE TO CONSOLE
     C
0004          CALL OPEN (6,'CON:')
     C
     C WRITE TITLE
     C
0005          WRITE (6,2)
0006 2        FORMAT (28X,'GRAPH OF SIN')
0007          TYPE
0008          TYPE
     C
     C SET PI AND -PI
     C
0009          PI=3.1415926
0010          MPI=-PI
     C
     C MAIN LOOP
     C
0011          DO 100 ANGLE=MPI,PI,.12
     C
     C FIGURE OUT WHICH ELEMENT IN ARRAY SHOULD BE SET TO *,
     C SIN RETURNS -1 TO 1 WHICH IS CONVERTED TO -35 TO 35
     C AND THEN OFFSET SO FINAL RANGE IS 1 TO 70
     C
0012          WHERE=SIN(ANGLE)*35+35
     C
     C FIGURE OUT HOW MUCH TO BLANK IN THE OUTPUT ARRAY
     C
0013          IBLANK=MAX0(35,WHERE)
     C
     C AND BLANK IT
     C
0014          DO 15 I=1,IBLANK
0015 15       LINE(I)=' '
     C
     C HMM... WHICH SIDE OF ZERO ARE WE ON?
     C
0016          IF (WHERE .GT. 35)  THEN
```

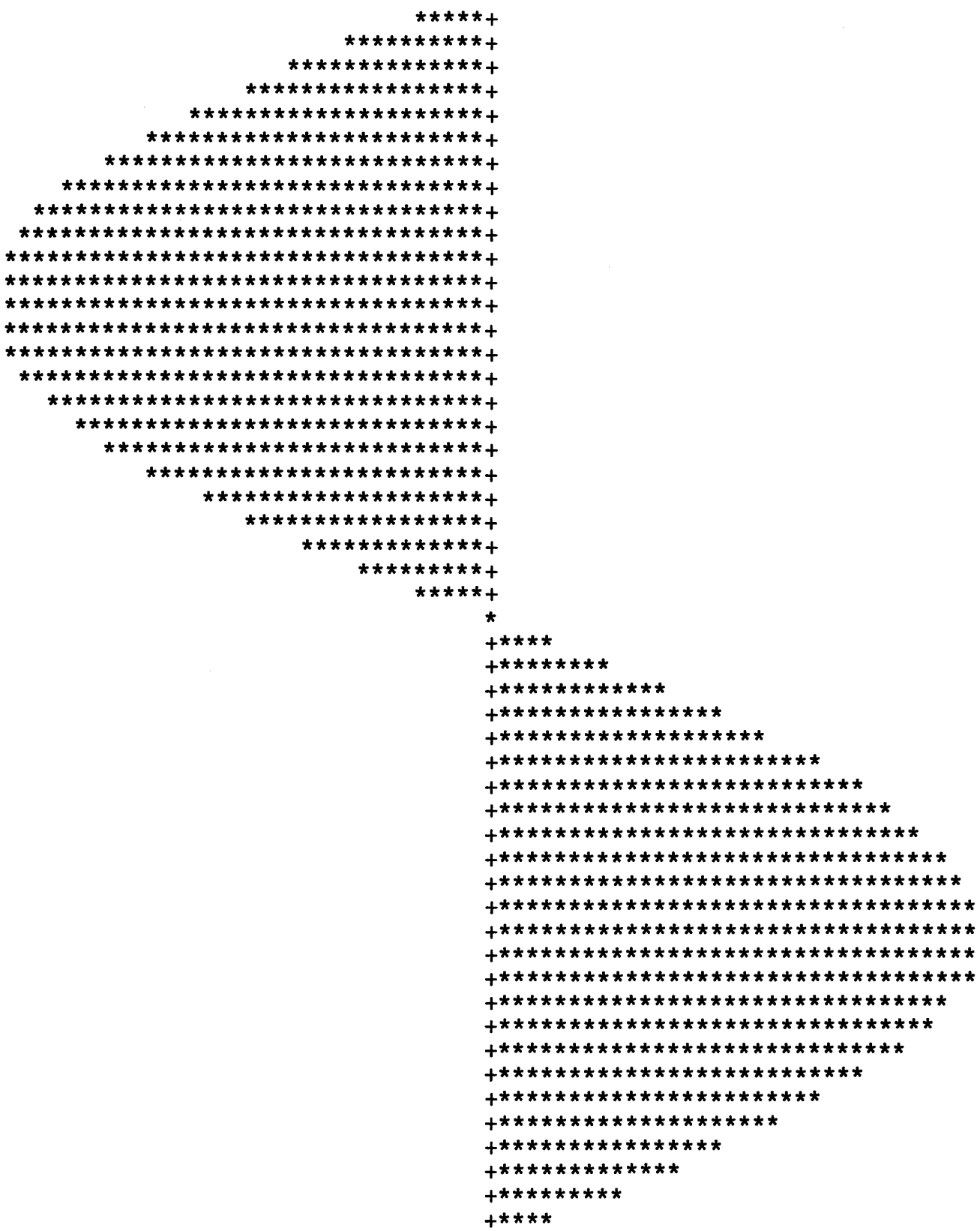
```
     C
     C RIGHT SIDE
     C
0017                          DO 20 I=36,WHERE
0018 20                       LINE(I)='*'
0019                              ELSE
     C
     C LEFT SIDE
     C
0020                          DO 30 I=WHERE,35
0021 30                       LINE(I)='*'
0022                              ENDIF
     C
     C SET "ZERO"
     C
0023          LINE(35)='+'
     C
     C AND THE SIN VALUE
     C
0024          LINE(WHERE)='*'
     C
     C IF THIS VALUE IS < 35, SET SO WE OUTPUT TO ZERO LINE
     C
0025          IF (WHERE .LE. 35)WHERE=35
     C
     C AND FINALLY OUTPUT THE LINE
     C
0026          WRITE (6,21) (LINE(I),I=1,WHERE)
0027 21       FORMAT (70A1)
0028 100      CONTINUE
0029          CALL EXIT
0030          END
** Generated Code =   687 (Decimal), 02AF (Hex) Bytes
** Array Area     =   420 (Decimal), 01A4 (Hex) Bytes

  No Compile errors


NO ASSEMBLY ERRORS.                    175 LABELS WERE DEFINED.
```

B>FRUN GRAPH {execute the program}

GRAPH OF SIN



```
B>FORT LOAD.X       {compile, listing to console, .ASM
                     and .OBJ to default drive}

NEVADA FORTRAN 2.00 (MOD 0)
Copyright (C) 1979, 1980, 1981, 1982 Ian Kettleborough

***** NEVADA Fortran 2.00 (Mod 0) ** Compiling File: LOAD.FOR *****

0001 C
     C "LOAD.FOR"
     C
     C THIS ROUTINE DEMONSTRATED THE USE OF THE 'LOAD' ROUTINE
     C TO LOAD AN ASSEMBLY LANGUAGE FILE INTO MEMORY AND
     C THEN CALL IT FORM FORTRAN
     C
0002          INTEGER A
     C
     C FIND OUT WHICH ONE TO LOAD
     C
0003          TYPE 'Enter 0 to "LOAD" LD.HEX'
0004          TYPE 'Enter 1 to "LOAD" LD.OBJ'
0005          ACCEPT 'Which one: ',LTYPE
     C
0006          IF (LTYPE .EQ. 0) THEN
0007                            TYPE '"LOAD"ing "LD.HEX"'
0008                             ELSE
0009                            TYPE '"LOAD"ing "LD.OBJ"'
0010                             ENDIF
     C
     C MUST LOAD "LD.HEX" OR "LD.OBJ" INTO MEMORY
     C BEFORE WE CAN CALL IT
     C
0011          CALL LOAD ('LD',LTYPE,IER)
     C
0012          TYPE 'ERROR FOR LOAD=',IER
     C
     C CHECK THE RETURNED ERROR CODE FROM LOAD
     C
0013          IF (IER .NE. 0)STOP 'LOAD ERROR'
     C
     C "CALL" THE ROUTINE
     C
0014          A=CALL ($8000,1)
     C
     C RESULT SHOULD BE 2
     C
0015           TYPE  'THE RESULT OF THE ASSEMBLY ROUTINE  IS:
',A
0016          CALL EXIT
0017          END
** Generated Code =   405 (Decimal), 0195 (Hex) Bytes


 NO ASSEMBLY ERRORS.                    135 LABELS WERE DEFINED.
```

```
B>FRUN LOAD          {execute the program}

Enter 0 to "LOAD" LD.HEX
Enter 1 to "LOAD" LD.OBJ
Which one: 0
"LOAD"ing "LD.HEX"
ERROR FOR LOAD=          0
THE RESULT OF THE ASSEMBLY ROUTINE IS:          2
```

```
B>FORT TRACE.ZCC    {compile with no listing, .ASM and
                     .OBJ to drive C}


B>FRUN TRACE        {execute the program}
STARTING EXECUTION
#: 4
          1
          2
          3
          4
#: 140
Pgm  is executing line 0009 in routine MAIN
Pgm  is executing line 0010 in routine MAIN
          1
Pgm  is executing line 0010 in routine MAIN
Pgm  is executing line 0014 in routine MAIN
ERROR TRAPPED, IER=          23
STOP  END IN - MAIN
```

```
B>FORT SORT.BBB     {compile with listing, .ASM and
                     .OBJ files to drive B}


B>FRUN SORT         {execute the program}
Shell sort

How many numbers (2-2000) 100
Starting sort
D=         50
D=         25
D=         13
D=          7
D=          4
D=          2
D=          1
All done

          2          6          6          6          7          8
          8          9          9         12         12         14
         14         18         19         21         21         21
         21         22         25         27         28         28
         29         29         30         32         33         33
         34         34         37         38         39         40
         40         41         46         46         48         48
         48         49         50         51         52         54
         58         58         59         60         60         61
         62         62         64         64         65         65
         66         67         68         70         70         71
         73         74         75         76         76         77
         80         80         82         82         83         84
         85         86         88         89         89         91
         91         91         93         93         93         93
         94         96         96         96         97         98
         99         99        100        100
How many numbers (2-2000) 0
STOP
b>
```

11.13.0 SUGGESTED FURTHER READING

SOFTWARE TOOLS
Brian W. Kernigham and P. J. Plauger
Addison-Wesley Publishing Co. 1976

A GUIDE TO FORTRAN PROGRAMMING
Daniel D. McCracken
Addison-Wesley Publishing Co. 1961

FORTRAN IV WITH WATFOR AND WATFIV
Cress, Dirksen, and Graham
Prentice-Hall, Inc. 1970

FORTRAN IV
Elliot I. Organick and Loren P. Meissner
Addision-Wesley Publishing Co. 1966

PROGRAMMING PROVERBS FOR FORTRAN PROGRAMMERS
Henry F. Ledgard
Hayden, 1975

ELLIS COMPUTING, NEVADA COBOL Application Package Bookl,
ELLIS COMPUTING, 1980

ELLIS COMPUTING, NEVADA EDIT, ELLIS COMPUTING, 1982.

ELLIS COMPUTING, NEVADA SORT, ELLIS COMPUTING, 1982.

ELLIS COMPUTING, NEVADA COBOL, ELLIS COMPUTING, 1979.

Starkweather, J., NEVADA PILOT, ELLIS COMPUTING, 1982.