PHILIPS

P855/P860

Full FORTRAN IV

second draft

data systems

document screening record

PHILIPS data systems

from: manual writing small computers
to:

date:

Please read the attached .....
Document name: .....
and give your comments in respect of .....

comments:

If this form is not returned to the above Department on or before ....., we shall assume that you do not wish to comment, and shall proceed accordingly.

The FORTRAN IV language described in this manual is designed for the P855/P860 computers with a minimum of 12K memory, and is intended to comply with the FORTRAN IV specification laid down by the American Standards Association. A copy of the ASA Full FORTRAN IV is given in the Appendix, but is included solely for reference purposes so that programmers who are interested may make a comparison between P855/P860 Full FORTRAN IV and that of the ASA; it is not intended to be used as a programming manual for the P855/P860

Great care has been taken to ensure that the information contained in this handbook is accurate and complete. However, should any errors or omissions be discovered, or should any user wish to make a suggestion for improving this handbook, he is invited to write his comments on the sheet provided at the end of the book and send it to:

NV Philips Electrologica
Industry Group Small Computers
P.P. Box 245
Apeldoorn
The Netherlands.

The Full FORTRAN IV compiler accepts FORTRAN IV source programs as input and produces object modules to be processed by the Linkage Editor/Loader with the FORTRAN System Library. The result of the link-load or link-edit process is a self-contained executable FORTRAN program which can run under the control of any Monitor.

The compiler is self-initializing and does not require reloading between successive compilations.

The system may accept source input from any peripheral device supported by the Monitor used.

In addition to the P855/P860 FORTRAN IV language elements, this manual contains information on the operating and control procedures, punching conventions for input from paper tape or from the operator's typewriter, and a comprehensive list of error messages.

Full details of the P855/P860 Monitors, their control commands and operating procedures can be found in the manuals on the Basic and Disc Monitors.

All programs consist of a number of characters grouped into lines and statements.
The P855/P860 FORTRAN IV language elements (variable names, constants, identifiers etc.) are constructed from the following characters:

(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
(digits) 0 1 2 3 4 5 6 7 8 9
(special signs)

=	Equals
	Blank
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Decimal point (period)
'	Apostrophe
$	Currency symbol

In place of the nh descriptor, Hollerith data may be enclosed within apostrophes.
An apostrophe within the character string must be coded as a double apostrophe.

LINE FORMAT

Each line consists of a Statement Label Field, a Line Continuation Field, a Statement Field and an Identification Field.

Statement Label Field (card columns 1-5)

Statements may be given a label so that they can be referred to in other statements. A statement label consists of from 1 to 5 digits and may be placed anywhere in the Statement Label field of the initial line of the statement; no particular order of numbers is required, and blanks and leading zeros are ignored. A particular statement label must only be used once in a program unit.
Column 1 of any line may contain the character X which is used for conditional compilation.

Line Continuation Field (card column 6)

A statement may be continued on an unlimited number of lines, provided that the continuation lines contain any character other than a blank or zero in column 6, and do not contain the character C (which denotes a Comment Line) in column 1.

Statement Field (card columns 7-72)

Any arithmetic, specification, control, Input/Output or function statement may be written in the statement field. Except in Hollerith data where they are significant characters, blanks may be used freely as they are ignored by the compiler.

Identification Field (card columns 73-80)

The last eight columns of a line are reserved for card sequence numbers. This field is ignored by the compiler, and may be left blank if the programmer wishes.

Comment Line

The letter C in column 1 of a line indicates a comment line. This has no effect upon the program itself and is solely for the programmer's convenience.
A comment line may be written anywhere in the program text.

Initial Line

An initial line is the first line of a statement, and contains a zero or blank in column 6 and an optional statement label or blanks in columns 1-5.

The data types defined for P855/P860 Full FORTRAN are integer, real, double precision, complex, logical and Hollerith.

Integer

An integer is internally represented by one 16-bit word. The range of an integer is $-32767 \leq i \leq +32767$ (2^{15-1}).

Real

A real number is represented in floating point format, consisting of a two word mantissa followed by a one word exponent.
The mantissa is a 31-bit binary fraction (bit 0 of the second word is not used) which is normalized so that the binary point is assumed to be between bits 0 and 1 of the first word. The range of the mantissa, m, is

$-1 \leq m < -\frac{1}{2}$ and $\frac{1}{2} \leq m < 1$

The binary exponent is an integer.

The range of a real number is

$-2^{2^{15}-1} \leq$ real number $\leq + 2^{2^{15}-1}$ $(|r| \leq 10^{9868})$

with an accuracy of 8 or 9 decimal digits.

Double Precision

A double precision number occupies 4 words of memory, consisting of a 46 bit mantissa (bit 0 of the second and third words is unused), normalized in the same way as a single precision number, and a 16-bit exponent. The range of values corresponds to real type but with an accuracy of 12 to 13 decimal digits.

Complex

A complex number occupies 6 words of memory and is formed by two consecutive real numbers.

Logical

A logical entity may only assume values of TRUE or FALSE, and is contained in one 16 bit word. The logical value TRUE has an internal value of -1 (all sixteen bits set to 1), and FALSE has a value of 0 (all bits 0).

Hollerith

Hollerith data are written as a string of ASCII characters.

CONSTANTS

A constant is an explicit numeric value which cannot be re-defined.

Integer Constant An integer constant is written as a string of digits, the maximum value of which may not exceed 32767, and whose minimum value may not be less than -32767 It may not contain a decimal point in any position in the string. If the number is positive, the preceding plus sign is optional.

Examples

59 +59 -95 2345 +6 -8000

Real Constant A basic real constant is written as a digit string which is the integer part, followed by a decimal point and a further digit string representing the decimal fraction part (either the integer or decimal fraction part may be empty, but not both). This may be followed by a decimal exponent which is written as the letter E followed by an optionally signed integer constant indicating the power of ten (positive or negative) by which the original constant is to be muptiplied.
A real constant is written as a basic real constant, a basic real constant followed by a decimal exponent, or an integer constant followed by a decimal exponent.

Examples

-65.09 8.0E+15 (equivalent to 8.0×10^{15}) +9.17

1000E-1 .7 90. -.5

Double Precision Constant A double precision constant has the same format as a real constant except that the letter D is used in place of the letter E in the exponent part.

Examples

2.35D1 -1111D2 57.62D-5 8D+12

Complex Constant A complex constant is formed by two optionally signed real constants which are separated by a comma and enclosed with parentheses. The first real constant of the pair represents the real part of the complex constant, the second real constant represents the imaginary part.

Examples (7.0, 12.76) (-18.1, -2.32) (0.53, 9.2) (+22.2, +18.5)

Logical Constant Logical constants are written as .TRUE. and .FALSE.; .TRUE. has an internal value of -1, .FALSE. has an internal value of 0. These are the only values that may be assumed.

Hollerith Constant A Hollerith constant is formed from an integer constant n, followed by the letter H, followed by exactly n ASCII characters. Blanks are significant characters within a Hollerith string. This constant is used only in CALL and data initialization statements.

Examples

4HDATA 6HSTRING

Hexadecimal Constant A hexadecimal constant is written as the character $ followed by up to four hexadecimal digits.
A hexadecimal digit is one of the letters A,B,C,D,E or F, or any numeric digit.
Hexadecimal constants may be used in place of integer constants

VARIABLES

A variable is represented by a symbolic name consisting of one to six alphanumeric characters (the first of which must be a letter) which represents a quantity whose value may be defined and re-defined several times during execution of a program.

The type of the variable (integer or real) is determined by the first letter of the variable name. An intial letter of I,J,K,L,M or N indicates an integer variable, any other initial letter indicates a real variable. This convention can be overridden by using Type statements to specify variables as being of a particular type (REAL, INTEGER, DOUBLE PRECISION, COMPLEX or LOGICAL). In this case, the variable names are not bound by the implicit type specification.

Integer variables: NUMBER LIST INTVAR

Real variables: ARRAY VECTOR ROOTS

Arrays

A variable name may represent a list or array of data instead of a single quantity. A particular element of an array is denoted by the array name followed by a subscript enclosed in parentheses. A subscript may be a single subscript expression or a list of expressions separated by commas. The number of subscript expressions must correspond to the number of dimensions declared for that array, and may not be greater than three.

The evaluation of an expression determines the array element being referenced. Any expression which has a positive, integer value is acceptable.

Examples

```
X(10)
ABC(3*N+6, K-5, 2)
LEMON (I)
```

An array declarator specifies an array used in a program unit, and indicates the symbolic name, the number of dimensions, and the size of each of the dimensions.

An array may be declared in a Type statement, a DIMENSION or a COMMON statement.

An expression is defined as any combination of constants, variables array elements, function references and operation signs.
There are three types of expressions: arithmetic, logical and relational.

ARITHMETIC EXPRESSIONS

An arithmetic expression is formed with operators and elements and defines a numerical value.

The arithmetic operators are:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	**

These are the only mathematical operations allowed; any others must be built up from these or computed by using the standard functions or subroutines available.

An element in an arithmetic expression is an array element, function reference, constant or variable which may be integer, real, double precision or complex.

Arithmetic expressions are evaluated according to certain rules of precedence, these being

**	highest priority
* and /	
+ and -	lowest priority

Where operators have the same level of priority, evaluation is from left to right.
Parentheses are used to group expressions and to alter the normal order of operations; the inner operation will be evaluated first, as in mathematical notation. Thus $(X+Y)^3$ must be written (X+Y)**3; X+Y**3 is a valid expression but does not have the same meaning and will not produce the desired result.

Using either parentheses or the rules of precedence, the following expressions are equivalent:

```
X*Y + A/C - K**N
(X*Y) + (A/C) - (K**N)
```

Two operators must never be adjacent. Thus A*-B is not a valid expression, although A*(-B) is.

In this version of Full FORTRAN IV, the elements of an arithmetic expression may be mixed mode. The results of mixed mode expressions are shown below.

For + - / * operations:

1st operand \ 2nd operand	I	R	D	C
I	I	R	D	C
R	R	R	D	C
D	D	D	D	X
C	C	C	X	C

For ** operations:

1st operand \ 2nd operand	I	R	D	C
I	I	R	D	X
R	R	R	D	X
D	D	D	D	X
C	C	X	X	X

where I = Integer (or logical)
R = Real
D = Double Precision
C = Complex
X = Illegal

RELATIONAL EXPRESSIONS

A relational expression consists of two arithmetic expressions separated by a relational operator, The value of the expression will be TRUE or FALSE depending on the truth of the relation described. The arithmetic expressions may be of any type except complex; one expression may be real or double precision, the other may be either of these, or both may be integer.
The relational operators are:

.LT.	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to
.GT.	greater than
.GE.	greater than or equal to

The period on either side of the operators is necessary to distinguish them from variable names used by the programmer.

Examples

6 .GT. 1	has a value of TRUE
5 .LT. 1	has a value of FALSE
M .LE. 8	is either TRUE or FALSE depending on the value of the variable M.

LOGICAL EXPRESSIONS

A logical expression is formed with logical operators and logical elements and has a value of TRUE or FALSE. The logical operators are:

.XOR.	exclusive or	(P not equivalent to Q. Modulo 2 sum)
.OR.	logical disjunction	
.AND.	logical conjunction	
.NOT.	logical negation	

Logical elements are:
array elements, function references, constants, variables or relational expressions. A logical variable or array must be declared in a LOGICAL type statement.

The result of the evaluation of a logical expression is internally represented by one 16-bit word; if bit 0 of the result is 1, the logical value of the expression is assumed to be true, and if it is 0, the logical value of the expression is assumed to be false.

Any integer-valued arithmetic expression may be used instead of a logical expression in any statement or expression, and conversely, any logical expression may be used instead of an integer expression.

The table below shows the results of evaluation.

Function	Bit i of first operand	Bit i of second operand	Bit i of result
.NOT.	0	-	1
	1	-	0
.AND.	0	0	0
	0	1	0
	1	0	0
	1	1	1
.OR.	0	0	0
	0	1	1
	1	0	1
	1	1	1
.XOR.	0	0	0
	0	1	1
	1	0	1
	1	1	0

This provides FORTRAN with the same bit handling possibilities as Assembly language, for example,
to set bit 3 of the variable L:

```
L = L .OR. $'1000
```

to change the parity of variable M:

```
M = M .XOR. 1
```

to make a branch to statement 500 if bit I is set in the variable K:

```
IF ((K .AND. (2** (15-I))) .NE. 0) GOTO 500
```

Relational and logical operators are subject to a priority order-which can be changed, as in mathematical notation, by the use of parentheses.

The overall order of precedence is

operator	priority
**	1st (highest)
* /	2nd
+ -	3rd
.LT. .LE. .EQ. .NE. .GE. .GT.	4th
.NOT.	5th
.AND.	6th
.OR.	7th
.XOR.	8th

If the values of the logical variables A and B are FALSE and TRUE respectively, the following expressions will have different results:

```
.NOT. (A .OR. B)
.NOT. A .OR. B
```

In the first expression, A .OR. B is evaluated and the result is TRUE, but .NOT. (TRUE) is FALSE, which is the value of the expression.

In the second expression, .NOT. A is evaluated first, the result of which is TRUE, but TRUE .OR. B implies TRUE which is the value of the expression.

Examples of logical expressions

```
L .AND. .NOT. (I .GT. F)
.NOT. W .AND. .NOT. Y
(ROOT ** F .GT. F) .AND. W
.NOT. (-X .GE. Y)
```

Combinations of Logical operators (.AND. .NOT., .OR. .NOT.) may be written next to each other, but this is the only instance where operators of the same type may be written together without requiring parentheses.

ASSIGNMENT STATEMENTS.

Statements may be classified as being executable or non-executable. Executable statements specify actions to be performed and result in object code output; non-executable statements describe the characteristics and arrangement of data etc., and do not have any corresponding object code instructions.
Assignment statements are executable and are of three types:

Arithmetic assignment statement,
Logical assignment statement,
GO TO assignment statement.

ARITHMETIC ASSIGNMENT STATEMENT.

An arithmetic assignment statement is written in the form

v = e

where v is a variable or array element identifier which may be of any type other than logical, and e is an arithmetic expression. After execution of the arithmetic expression its computed value is assigned to the variable on the left of the equals (=) sign. This statement is very similar to an algebraic equation except that, in FORTRAN, the equal sign means 'is to be replaced by' rather than 'is equivalent to'.
The special rules for assignment are shown in the table following:

If v is	And e is	Assignment rule is
Integer	Integer	Assign
Integer	Real	Fix and Assign
Integer	Double precision	Fix and Assign
Integer	Complex	Combination not permitted
Real	Integer	Float and Assign
Real	Real	Assign
Real	Double precision	DP Evaluate and Real Assign
Real	Complex	Combination not permitted

Double Precision	Integer	DP Float and Assign
Double Precision	Real	DP Evaluate and Assign
Double Precision	Double Precision	Assign
Double Precision	Complex	Combination not permitted
Complex	Integer	Combination not permitted
Complex	Real	Combination not permitted
Complex	Double Precision	Combination not permitted
Complex	Complex	Assign

Assign means transmit the resulting value, without change, to the variable.
Fix means truncate any fractional part of the result and change that value to an integer.
Float means alter the value to a real number.
DP Evaluate means evaluate the expression, using double precision computation.
DP Float means change the value to a double precision number, retaining the greatest precision of the value that can be held in a double precision entity.

Examples:

A = B	The value of A is replaced by the current value of B.
I = X	The value of X is truncated to an integer value, and this value then replaces the value of I.
N = N + 5	The value of I is replaced by the value of I + 5.
R = M ** N	M is raised to the power N and the result is converted to a real value which replaces the value of R.

LOGICAL ASSIGNMENT STATEMENT

A logical assignment statement is written in the form

v = e

where v is a logical variable or logical array element, and e is a logical expression. The computed value of the expression is assigned to the logical variable.

Examples

X = .TRUE. The value of X is replaced by the logical constant .TRUE.

LVAR = 5 .GT. J If the integer constant 5 is greater than the value of J, LVAR will be assigned the logical constant .TRUE. If 5 is not greater than J, LVAR is assigned the value .FALSE.

The following example is given to illustrate the differences that may arise because of the P855/P860 FORTRAN extension to logical expressions (i.e. that any integer-valued expression may be used instead of a logical expression, and vice-versa).

LOGICV = .NOT. A

If A were solely a logical variable, and the value of A were TRUE, the value of LOGICV would be replaced by the logical constant .FALSE.; if A were FALSE, the value of LOGICV would be replaced by .TRUE.

However, because of the extension, A may be a logical or an integer variable (one 16 bit word in memory). This variable is considered to be TRUE if bit 0 is 1, otherwise it is taken to be FALSE. The logical operator .NOT. changes all 16 bits (see the rule on page 11) and the result is stored in LOGICV.

Thus, if A were TRUE (because bit 0 is 1, irrespective of the other bits), LOGICV would be taken to be FALSE (bit 1 would be 0) but this does not necessarily mean that LOGICV would have the logical constant value .FALSE. (where all 16 bits would be 0).

If LOGICV is used in a logical expression (in a logical IF, for example) the result will be the same as if the constant .FALSE. had been assigned to LOGICV, but if it is used as an arithmetic expression the result may be quite different.

In both Arithmetic and Logical Assignment statements the = sign must be written on the initial line of the statement.

GO TO ASSIGNMENT STATEMENT

This statement is written

ASSIGN k TO i

where k is a statement label and i is an integer variable. After such an assignment, any subsequent execution of an assigned GO TO statement which uses the integer variable (i) will cause the statement which is identified by the assigned statement label to be executed next, unless the variable has been redefined.

The statement label must refer to an executable statement in the same program unit in which the ASSIGN statement appears.

A variable used in an ASSIGN statement may not be referenced in any other type of statement unless it is redefined.

Examples

```
          ASSIGN 150 TO NUMBER
100       GO TO NUMBER, (15, 125, 90, 150, 175)
          .
          .
          .
150       X = Y**Z
```

Statement 150 will be executed immediately after statement 100.

```
          ASSIGN 25 TO INDEX
          .
          .
10        GO TO INDEX, (5, 15, 25, 50)
          .
          .
5         N = I**J
          .
          .
25        I = L-M
          ASSIGN 15 TO INDEX
          GO TO 10
          .
          .
15        X = F*R+W
```

In the above example, the first time statement 10 is executed control is transferred to statement 25. On the second execution of statement 13, control is transferred to statement 15.

Program statements are usually executed sequentially, but it is often necessary to alter the order of execution - sometimes a section of program has to be executed repeatedly, using different sets of data, or a branch may be made to a particular section of a program, depending on the values of computed results.
This section describes the statements that may be used to alter and control the normal order of execution.

In any control statement the keyword (CALL, GO TO, IF etc.) must be written completely on the initial line of the statement.

There are nine types of control statements:

1. GO TO statements
2. Arithmetic IF statement
3. Logical IF statement
4. DO statement
5. CONTINUE statement
6. CALL statement
7. RETURN statement
8. STOP and PAUSE statements
9. END statement

GO TO STATEMENTS.

There are three types:
- Unconditional GO TO,
- Assigned GO TO,
- Computed GO TO.

Unconditional GO TO

This statement is written

GØ TØ n

where n is a statement label.
When this statement is encountered, the next statement executed will be the one whose label is specified in the GO TO.

Example:

```
    2   I = K ** 2
        GØ TØ 10
        .
        .
   10   R = SQRT (Z + Y)
        .
        .
```

Assigned GO TO statement

This is written

$$GØ\ TØ\ i, (k_1, k_2, \ldots, k_n)$$

where i is an integer variable and the k's are statement labels.

When executing an assigned GO TO statement, the current value of i must have been assigned to be one of the k_n statement labels (by the execution of an ASSIGN statement). Execution of the assigned GO TO causes the statement whose label (k) is the value of the variable i to be executed next.

Example:

```
      ASSIGN 15 TØ I
      .
      .
20    GØ TO I, (50, 10, 15, 3)
15    X = Y - Z
      ASSIGN 50 TO I
      GO TO 20
      .
      .
20    Z = Y ** 3
      .
      .
```

Computed GO TO statement.

This statement is written

GØ TØ (k_1, k_2, ... , k_n), i

where the k's are statement labels and i is a non-subscripted integer variable.

This is a multiple-branch statement where the value of the integer variable determines which statement within the parenthesized list will be executed next. Control is transferred to the statement labelled k_1, k_2, ... or k_n, depending on whether the current value of i is 1, 2, 3, ... , or n, respectively. That is, if the variable has a value of 1, the first label in the list is the label of the next statement to be executed. If i is greater than n, an error message will be generated.

Example:

GØ TØ (9, 7, 5, 3, 1), LABEL

If LABEL has a value of 5, the next statement executed is that which is labelled 1.

If LABEL has a value of 1, the statement labelled 9 will be executed next.

ARITHMETIC IF STATEMENT

This has the format

IF (e) k_1, k_2, k_3

where e is an arithmetic expression of any type except complex, and the k's are statement labels.
If the value of the expression within parentheses is negative, the statement which has the label n_1 will be executed next; if the value of the expression is zero, statement n_2 will be executed next; and if the expression has a positive value n_3 is the next statement to be executed.

Example:

```
    IF (A - Y ** 2) 10, 20, 20
10  A = Y ** 2 + A ** 2
    GØ TØ 15
20  Y = (X + C) * B
15  CONTINUE
```

If the computed value of the expression is negative, control will be transferred to the statement labelled 10; if the expression has either a zero or a positive value, the next statement executed will be the one labelled 20.

LOGICAL IF STATEMENT

This is written

IF (e) S

where e is a logical expression and S is any executable statement except a DO or another logical IF.
If the value of the expression is true, statement S will be executed, followed by the next statement in the written sequence. If the expression if false, S will be ignored and the program continues with the next statement.
In both cases, the next statement to be executed is the one written after the logical IF, unless S is a GO TO or an arithmetic IF statement and the expression is true.

Examples:

```
    5  IF (W.NE.A ** B) W = 0.0
   10  X = A * Z
```

If the expression is true, W = 0.0 will be executed, followed by the statement with the label 10.
If the expression is false, W = 0.0 will be ignored and statement 10 will be executed.

```
    5  IF (B.LE.5.75) GO TO 20
   10  NUMBER = INT * LIST
   12  IF (B.EQ.C) ANSWER = 7.0 * (B/A)
   15  A = ANSWER + NUMBER
   20  B = A ** 2
```

If the value of the expression in the statement labelled 5 is true (i.e. B is less than or equal to 5.75), GO TO 20 is executed and control is passed to the statement labelled 20. If the expression is false, the statement GO TO 20 is ignored and the next statement executed is 10.

DO STATEMENT

The DO Statement may be written in either of two forms:

DØ n i = m_1, m_2, m_3

or DØ n i = m_1, m_2

where n is a statement label, i is an integer variable (called the control variable), and m_1, m_2, and m_3 are each either an integer constant or a non-subscripted integer variable reference (of which only m_3 must have a value greater than zero).

m_1 is known as the initial value, m_2 the terminal value, and m_3 the incrementation value. If m_3 is not specified, an increment of 1 is assumed.

The DO statement is a command to repeatedly execute the successive sequence of statements up to and including the statement labelled n.

That is, all the statements following the DO up to statement n represent a loop. For the first pass through the loop, the statements are executed with i = m_1; at each succeeding execution i is increased by m_3 (or 1 where m_3 is not specified). If the value of the control variable is less than or equal to the terminal value, the range of the DO (i.e. the sequence of statements comprising the loop) is executed again until the value of the control variable becomes greater than the terminal value.

At this point the DO is satisfied, the control variable becomes undefined and control passes to the statement following the one labelled n.

The terminal statement in a DO-loop must not be a GO TO statement of any type, an arithmetic IF, a RETURN, STOP, PAUSE, another DO statement, or a logical IF containing any of these.

The comma following the initial parameter in a DO statement must appear on the initial line of the statement.

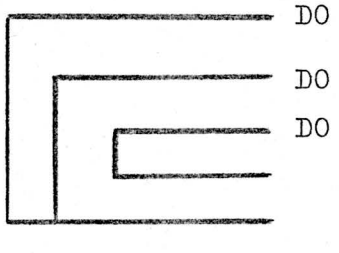
A conditional branching statement may cause control to pass out of the DO-loop before the DO is completely cycled. When this occurs, the current value of the control variable i remains defined.

In this version of FORTRAN, the range of the DO may contain statements which cause the value of any of the parameters (the control variable i, m_1, m_2, and m_3) to be redefined. That is, any of these quantities may appear on the left of an equal sign in an arithmetic statement.

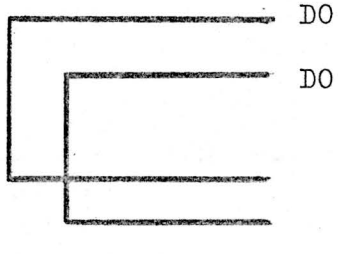
Transfer of control from inside the range of a DO to outside its range is permitted at any time. The reverse, however, is not permissible and control cannot be passed to any statement within the range without first executing the DO statement itself. (However, no check is peformed by the compiler that such an error has occurred, and no diagnostic can be issued).

The range of a DO-loop may contain other DO's. When this occurs (known as nesting), all statments in the ranges of the inner DO's must be completely contained by the outer loop, although the terminal statement may be common to two or more DO loops.

Examples:



This is a permissible combination



This is not permissible as the range of the inner DO extends beyond that of the outer loop.

```
STATEMENT
1    5 6 7   12   16   20   24   28   32   36   40   44   48   52   56   60   64   68
      DO 50 N=1,100
50    ISTOCK(N)=ISTOCK(N)-IOUT(N)
51
```

Statement 45 is a command to execute the statements following, up to and including statement 50. The first time through the loop N will have a value of 1; at each succeeding execution its value will be increased by 1 until it exceeds 100; control then passes to the statement labelled 51.

```
      DO 10 I=1,10
      DO 10 J=1,10
      IF(A(I)-B(J)) 10,20,10
10    CONTINUE
      GOTO 50
20    LOC(1)=I
      LOC(2)=J
50    CONTINUE
```

The example above looks for the first duplication of values in two arrays, and records this index in the arrays. The IF statement may cause a 'special exit' from the inner loop. When this happens, the index values of all loops are preserved.

CONTINUE STATEMENT

This is written

CONTINUE

This statement is a dummy statement which provides a common finishing point for a DO-loop, and causes the normal sequence of execution to be continued.

CALL STATEMENT

A CALL statement has the form

CALL s (a_1, a_2, ... , a_n)

or CALL s

where s is a subroutine name and the a's are actual arguments. The CALL statement transfers control to the subroutine subprogram and associates the dummy parameters in the subroutine with the actual arguments that appear in the CALL statement.

An actual argument may be a constant (including Hollerith) variable, array name, array element name, an expression, or the name of a Function procedure or another subroutine.

The actual arguments must agree in order, number and type with the corresponding dummy arguments in the subroutine.

RETURN STATEMENT

This is of the form

RETURN

This statement returns any computed value and control to the calling program unit from a function subprogram or a subroutine.

STOP and PAUSE STATEMENTS

These are both program control statements and have the effect of halting execution of the running program (the compilation process is not affected).

The STOP statement is written

STOP or STOP n

n is a string of up to four alphanumeric characters.
An = sign may not be included in the string.
This statment terminates execution of the object program.

The PAUSE statment is written

PAUSE or PAUSE n

where n is a string of up to four alphanumeric characters, indicating at which point in the program the halt is occurring. The machine waits waits until the operator types in

(LF) (CR)

causing execution to be resumed at the statement after the PAUSE.

When a halt occurs as a result of either of these statements, "F:" followed by the string n is output to the operator's typewriter, so that the programmer can easily see what stage of the program has been reached.

END STATEMENT

The END statement is written

END

and indicates the physical end of a program unit; an END statement in a main program causes control to be returned to the Monitor.

If a label is given to an END statement it will be ignored.

When an END statement is processed, a STOP statement is generated automatically, but this does not appear on the source listing.

Specification statements are non-executable in that they provide the compiler with information about the data in the source program, and do not result in the creation of any instructions in the object program. They also supply the information required to allocate storage locations for the data.

There are six types of specification statements:

- DIMENSION statement
- COMMON statement
- EQUIVALENCE statement
- EXTERNAL statement
- Type statements
- DATA initialization statement

DIMENSION STATEMENT

This statement has the form:

DIMENSION $v_1(i_1)$, $v_2(i_2)$,...,$v_n(i_n)$

where each v(i) is an array declarator;
v is the declarator name, and (i), the subscript, is composed of up to three expressions, separated by commas. The subscript expressions may be either integer constants or, in the case of adjustable dimensions, integer variable names. The values of the subscript expressions give the maximum size of each dimension.

The DIMENSION statement provides the information necessary to allocate storage for arrays used in the source program. The number of words reserved for an array depends on the array name (which implicitly specifies its type) or its declared type.

The number of locations reserved for an integer or Boolean array is the same as the number of elements in the array; three words are reserved for each element of a real array, four for each element of a double precision array, and six for elements of a complex array.

Example

```
DIMENSION JEST(200), EXP(2,5,25), XARRAY(2,50)
```

The elements of the integer array JEST would occupy 200 memory locations, real array EXP would occupy 2*5*25*3=750 locations, and real array XARRAY would occupy 2*50*3=300 locations.

Adjustable Dimensions

In the foregoing examples, the maximum value of each subscript in the declarator was specified by a numeric value which is absolute and may not be altered. However, if an array is used in a Function or subroutine subprogram, each of the array dimensions may be specified by an integer variable. Such variables, and the array name, must appear in the dummy argument list of the subprogram. When the subprogram is called the actual arguments that represent array dimensions in the calling program become associated with those integer variables that represent the adjustable dimensions in the dummy argument list of the subprogram. The dimensions of a dummy array appearing in a subprogram, may alter each time the subprogram is called.

The absolute dimensions of an array must be declared in a calling program. The adjustable dimensions in a subprogram must not exceed the absolute dimensions of the actual array.

An identifier which appears in a COMMON statement can not be used to identify an adjustable array in a subprogram.

Example

Calling program

```
DIMENSION B(5,15)
.
CALL RTINE(B,3,10,..)
```

Subprogram

```
SUBROUTINE RTINE(X,I,J,...)
DIMENSION X(I,J)
W=A+X(I,J)
```

The absolute dimensions of the array B are declared in the calling program. When the subroutine RTINE is called, the dummy argument X is assigned the array name B, and the values 3 and 10 assigned to the dummy arguments I and J.

COMMON STATEMENT

The COMMON statement provides a means of sharing memory space between subprogram variables and main program variables. Data may also be transferred between subprograms or modules of a program.
The COMMON statement assigns two variables in different subprograms, or in a main program and a subprogram, to the same memory location.
The format of the statement is

COMMON $/x_1/a_1/.../x_n/a_n$

where each a is a list of variable names, array names or array declarators (no dummy arguments are allowed), and each x is an optional common block name which must not be the same as any variable or array name.
If no block name is specified at the beginning of the statement, all the entities listed, up to the first specification of a block name (between slashes), are in blank common; or if two slashes appear with no block name between them, all entities following are also in blank common.

If one program contains the statement

```
COMMON ALPHA
```

and another program has

```
COMMON BETA
```

the variables ALPHA and BETA refer to the same storage location.

If a main program contains the statements

```
REAL P,Q,R
COMMON P,Q,R
```

and a subprogram contains the statements

```
REAL A,B,C
COMMON A,B,C
```

the result will be that P shares the same storage location as A, Q shares the same location as B, and R and C share the same location.

All the variables named in a COMMON statement are assigned to storage locations in the sequence in which the names appear in the statement.

The number of locations occupied by a common block depends on the number and mode of the elements that are introduced through COMMON and EQUIVALENCE statements. Labelled common blocks that have the same name in several modules

of one program must be the same size. The sizes of blank common blocks in program units that are to be executed together need not be the same.

Where an array is declared in a DIMENSION statement, the array name may be written, without the subscript, in a COMMON statement. However, where the array subscript is specified in a COMMON statement, the same information must not also be given in a DIMENSION statement.

EQUIVALENCE STATEMENT

The EQUIVALENCE statement causes two or more variables in one program unit to be assigned to the same memory location. The format of the statement is:

EQUIVALENCE (k_1), (k_2),...,(k_n)

where each k represents a list of two or more variables or array elements (subscript expressions must be positive, integer constants). Each element in a list is assigned the same area of memory by the compiler.

One application of the use of EQUILAVENCE is that the programmer can make use of the same location to contain variables which are quite different from each other but which are never needed at the same time. For example, a variable may appear in the input list and then be used only in the first few statements of the program; then, later in the program , a value may be assigned to a different variable to be used as a parameter in a DO list, and similarly with other variables at different stages in the program. Thus, locations would be allocated to these variables whose values would be required only at certain, seperate points within the program. By using EQUIVALENCE all these variables could be assigned to the same location.

Another use of this statement allows the programmer to establish equivalence between variables which have different names but which all mean the same thing. After writing a long program a programmer may discover that he has given different names to variables which are, in fact, the same. He could go back through the program and alter the names, but it is much simpler to list the names in an EQUIVALENCE statement.

However, the greatest value in the use of equivalence lies in establishing equivalence between arrays. When one array is made equivalent to another, an equivalence is established between other elements of their arrays

For example, the statement:

EQUIVALENCE (X(4), Y(2), Z(1))

will establish equivalences as follows:

```
X(1)
X(2)
X(3)   Y(1)            - share the same location
X(4)   Y(2)   Z(1)     - share the same location
X(5)   Y(3)   Z(2)     -   "    "   "      "
X(6)   Y(4)   Z(3)     --  "    "   "      "
.      .      .
.      .      .
```

Any array name which is listed in an EQUIVALENCE statement must have a subscript; it is not sufficient to write just the array name. The number of subscript expressions must correspond to the number of dimensions declared for that array. If the array element name has more than one subscript expression this refers to the position in the array in the same way as in an arithmetic statement. A special rule exists - the element successor rule - to determine where a given element is stored in the linear sequence of memory locations. By apllying this rule, a two- or three-dimensional array can be made equivalent to a one- dimensional array of the same length.

The rule is given in the table below.

Dimensions	Subscript declarator	Subscript	Subscript value
1	(A)	(a)	a
2	(A,B)	(a,b)	a+A.(b-1)
3	(A,B,C)	(a,b,c)	a+A.(b-1)+A.B.(c-1)

where A,B, and C are dimensions, declared in a DIMENSION, COMMON or Type statement, and a, b and c are the values of subscript expressions.

Examples

```
DIMENSION M(3,12), N(15)
EQUIVALENCE (M(2,9), N(10))
```

To determine which position element (2,9) is in this array we can apply the above rule. Thus, if element (1,1) is in the first position in the linear sequence, element (2,9) is

$2 + 3 . (9-1) = 26$

```
DIMENSION I(5), J(10,10), K(5,10,5)
EQUIVALENCE (L, I(1), J(5,3)), (K(5,10,2),M)
```

This EQUIVALENCE statement causes the variable L, the first variable in array I (i.e. I(1)), and the 25th. variable in array J (i.e. J(5,3)) to be assigned to the same storage location.

Variables that are listed in both COMMON and EQUIVALENCE statements may increase the size of the common block of storage. For example,

```
COMMON A, C
DIMENSION A(4), B(4)
EQUIVALENCE (A(3), B(1))
```

Without the EQUIVALENCE statement, the common block would consist of fifteen locations (three locations for each real entity) in the sequence:

A(1), A(2), A(3), A(4), C

With the EQUIVALENCE statement, the array B is brought into common and causes the following sequence of storage allocation:

```
A(1), A(2), A(3), A(4), C
            B(1), B(2), B(3), B(4)
```

The common block is extended by the further three locations required for the array element B(4). It is quite permissible to lengthen a common block in this way, but as arrays are stored in consecutive forward locations, a variable must not be made equivalent to another array variable in such a way that the array precedes the beginning of the common area.
For example, the following statement is illegal:

```
COMMON L,M,N
DIMENSION K(3)
EQUIVALENCE (M, K(3))
```

This would cause K(1) to precede L as follows:

```
   L,   M,    N
K(1), K(2),K(3)
```

Therefore, only forward lengthening - towards increasing memory addresses - is allowed.

Two variables which are listed in a common block must not also be made equivalent.

When using both COMMON and EQUIVALENCE statements the programmer needs to remember the number of locations allocated to the different data types; an integer occupies one word of memory, a real value occupies three, double precision four words, complex six words, and a logical value one word.

EXTERNAL STATEMENT

The name of a subroutine or function may be used as an argument in a subprogram call. When this happens, the function or subroutine name must be declared in the list of an EXTERNAL statement in the calling program so that it is distinguished from a variable name.
The statement is written

EXTERNAL $a_1, a_2, \ldots, a_n$

where each a is a subprogram name.

Example

Calling program

```
      .
      .
      EXTERNAL NUMSUM
      .
      CALL SUBR (L, NUMSUM, X)
```

Subprogram

```
      SUBROUTINE SUBR (INT, Y, ARG)
      IF (INT) 5,10,10
    5 RES=Y(INT, ARG**2)
      .
      .
   10 RETURN
      END
```

In the above example, the subprogram name NUMSUM is used as an argument in the subroutine SUBR. The subprogram name NUMSUM is passed to the dummy argument Y as are the values of L and X to INT and ARG respectively. The subprogram NUMSUM will be called and executed only if the value of INT is negative.

TYPE STATEMENTS

Type statements are used to declare variables, arrays or functions as being of a particular data type. This overrides the normal rule of integer variable or array names beginning with one of the letters I through N, and of real names beginning with any other character. The format of the statement is:

Type X_1, X_2,..,X_n

where Type is one of the declarators INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL, and each X is a variable name, array name, array declarator or a function name, which is to be of the specified data type. Array dimensions may be specified.

Examples

```
INTEGER ALPHA, BETA, R22(100)
REAL NUMBR, LIST, JUMPS
DOUBLE PRECISION DMODE, Z, AD
COMPLEX CFUN, ZLIST
LOGICAL TRUTH, LVARBL, LLIST
```

DATA INITIALIZATION STATEMENT

The DATA statement is used to compile data values into the object program from source program statements.
The statment is written:

DATA list1/d_1/,list2/d_2/,../,listn/d_n/

where 'list' contains the names of variables and array elements that are to be given values; the d's are corresponding lists of optionally signed constant values. If one value is to be assigned to successive variables in the list, that constant may be preceded by an integer constant specifying the repeat number, and an asterisk.
An implied DO is accepted in DATA statements, provided that all the parameters are integer constants.

Example

```
DATA A, B, C(2)/15.0, 1.75, 4.0/, INR, LABEL/2*100/,
     LOG, VAL/.TRUE., .FALSE./,(R(I),I=1,10)/1.0, 2*5.75, 7.23,
     11.1, 3*5.5, 2*10.0/
```

Values for labelled common block elements can only be defined in a block data subprogram (see next chapter).

It is often necessary to write a program in which a particular sequence of statements is to be computed several times, with different arguments for each calculation. To simplify the writing of such programs it is possible to write these frequently-used operations only once in a program and then to refer to them whenever necessary, or to store them on a library, to be incorporated in the load module at link-edit or link-load time. These operations are in the form of subprograms which may be a series of FORTRAN or assembly language statements constituting a complete program, external to the calling unit, or FORTRAN statements which are defined within the program that uses them.

There are five categories of subprograms:

- Statement functions
- Library functions
- Function subprograms
- Subroutines
- Block Data subprograms.

STATEMENT FUNCTIONS

A statement function is defined by a single statement (similar to an arithmetic assignment statement) within the program in which it is used. The function need be defined only once, and then may be referred to by name whenever it is required in that program. The name of a statement function must not be the same as any variable or array name in the same program unit, nor must it be listed in an EXTERNAL statement.
The format of a statement function is:

$$f(a_1,a_2,\ldots,a_n) = e$$

where f is the function name, the a's are dummy arguments, and e is any arithmetic expression (which may contain subscripted variables). This expression defines the computations which are to be performed when the function name is used in an arithmetic statement.

The type of the function is determined either by the first letter of the name (integer or real indication), or by specifying its name in an explicit type statement.

The dummy arguments are variables which represent the actual values passed to the statement function. The actual arguments must agree in order, number and type with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression of the same type as the corresponding dummy argument.

Apart from dummy arguments, the expression e may contain
- constants (other than Hollerith)
- variable references
- Library function refrences
- references to other statement functions (which must be pre-defined)
- external function references.

A statement function is executed by writing its name wherever the value of that function is required, and by substituting actual arguments for the dummy arguments in the statement function definition. Once the value of the function is computed, that value is made available to the expression containing the function reference.

All statement function defintions to be used in a program must precede the first executable program statements.

A statement function is a closed subprogram, i.e. it appears in a program only once and the object program transfers control to the subprogram each time the function name is used in an expression in that program.

Examples

```
SUM(A,B,C,D) = A+B+C+D
MEAN (I,J,K,L,M) = (I+J+K+L+M)/5
FUNC(Z) = B+C * P*Z
```

The statement function definition

```
SUM (A,B,C,D) = A+B+C+D
```

can be used elsewhere inthe program simply by writing the function name, for example:

```
Y = SUM(W,X,Y,Z) / SUM(P,Q,R,S)
```

If the following function were to be used several times in a program, the necessary definition for that function is as shown.

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

Statement function definition:

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

LIBRARY FUNCTIONS

Library functions are processor-defined, commonly -used mathematical subprograms which are stored on the FORTRAN System Library. These functions are available to all programs or program units that might require them, and they are used simply by writing the appropriate name where required, accompanied by the correct number (and type) of arguments.

Library functions are also closed subprograms, inserted only once into the object program, irrespective of how many times they are used.

The table below lists the P855/P860 Library functions.

Function name	Argument type	Function type	Number of arguments	Definition
EXP	Real	Real	1	e^a Exponential
DEXP	Double precision	Double precision	1	
CEXP	Complex	Complex	1	
ALOG	Real	Real	1	$\log_e(a)$ Natural logarithm
DLOG	Double precision	Double precision	1	
CLOG	Complex	Complex	1	
ALOG10	Real	Real	1	$\log_{10}(a)$ Common logarithm
DLOG10	Double	Double precision	1	
SIN	Real	Real	1	sin(a) Trigonometric sine
DSIN	Double precision	Double precision	1	
CSIN	Complex	Complex	1	
COS	Real	Real	1	cos(a) Trigonometric cosine
DCOS	Double precision	Double precision	1	
CCOS	Complex	Complex	1	
TANH	Real	Real	1	tanh(a) Hyperbolic tangent
SQRT	Real	Real	1	$(a)^{\frac{1}{2}}$ Square root
DSQRT	Double precision	Double precision	1	
CSQRT	Complex	Complex	1	

ATAN	Real	Real	1	arctan(a)	
DATAN	Double precision	Double precision	1		Arctangent
ATAN2	Real	Real	2	arctan(a1/a2)	
DATAN2	Double precision	Double precision	2		
CABS	Complex	Real	1	$\|x\|$	Modulus
ABS	Real	Real	1		
IABS	Integer	Integer	1	$\|x\|$	Absolute value
DABS	Double precision	Double precision	1		
AINT	Real	Real	1	sign of x	
INT	Real	Integer	1	times largest	
IDINT	Double precision	Integer	1	integer $\leq \|x\|$	Convert, truncate
AMOD	Real	Real	2	a1(moda2)	
MOD	Integer	Integer	2	defined as	
DMOD	Double precision	Double precision	2	a1-(a1/a2)a2	Remainder
AMAX0	Integer	Real			
AMAX1	Real	Real			
MAX0	Integer	Integer	$\geq$ 2	$\max(a_1,a_2,..,a_n)$	Choose largest value
MAX1	Real	Integer			
DMAX1	Double precision	Double precision			
AMIN0	Integer	Real			
AMIN1	Real	Real			
MIN0	Integer	Integer	$\geq$ 2	$\min(a_1,a_2,..,a_n)$	Choose smallest value
MIN1	Real	Integer			
DMIN1	Double precision	Double precision			
FLOAT	Integer	Real	1	Integer to real conversion	
IFIX	Real	Integer	1	Real to integer conversion; truncate	
SIGN	Real	Real	2	Sign of a2 times $\|a1\|$	
ISIGN	Integer	Integer	2		
DSIGN	Double precision	Double precision	2		
DIM	Real	Real	2	a1-Min(a1,a2)	Positive difference
IDIM	Integer	Integer	2		

SNGL	Double precision	Real	1	Obtain most significant part of double precision argument
REAL	Complex	Real	1	Obtain real part of complex argument
AIMAG	Complex	Real	1	Obtain imaginary part of complex argument
DBLE	Real	Double precision	1	Express single precision argument in double precision format
CMPLX	Real	Complex	2	a1 + a2 $\sqrt{-1}$ Express 2 real arguments in complex form
CONJG	Complex	Complex	1	Obtain conjugate of complex argument
IOSTAT	Integer	Integer	1	Test status after I/O operation

FUNCTION SUBPROGRAMS

A Function subprogram is a FORTRAN program consisting of any number of statements which form a complete program unit, independent of any program which uses it. A Function subprogram will be executed wherever its name (and arguments) appears in another program.

A Function name consists of up to six alphanumeric characters, the first of which must be a letter. If the Function name has not been declared as being a particular type, the first letter must be consistent with the implicit type rule, i.e. if the type of the Function is integer the initial letter must be I, J, K, L, M or N, and if real, any other letter.

The first statement of a Function subprogram must be

$$\text{FUNCTION X } (a_1, a_2, \ldots, a_n)$$

where X is the Function name, and the a's are dummy arguments which may be used as variable names, array names or other Function or Subroutine names.
The word 'FUNCTION' can be preceded by a REAL, INTEGER, DOUBLE PRECISION, COMPLEX or LOGICAL specification (e.g. REAL FUNCTION NEWS (RAG,X2)) if the naming convention is to be overridden.

Following the FUNCTION statement, the subprogram may contain any combination of statements (other than SUBROUTINE, BLOCK DATA, or another FUNCTION statement).

A Function subprogram may not be called recursively, i.e. it may not call itself, nor may two subprograms call each other (if Function A calls Function B, B may not call A) unless both are compiled with the dynamic core allocation option; "cross recursivity" is then possible.

The name of the Function must appear at least once in the subprogram as a variable on the left-hand side of an arithmetic assignment statement; as the subprogram name has a value associated with it which is returned to the calling program, the value must be assigned to the name within the subprogram. During execution of the subprogram the variable name (of the Function) may be referenced or re-defined. The value of the variable at the time of execution of a RETURN statement in the subprogram, is then the value of the Function.

The arguments of a function subprogram may be referred to and redefined so that results may be returned to the calling program, in addition to the Function value.

As a Function subprogram is a complete program unit, it must contain an END statement wich indicates the physical end of the subprogram, and at least one RETURN statement which causes any computed value and control to be returned to the calling program unit.

Example

Calling program

```
      •
      •
      RESULT = COMP/FINE (N,P,R)
      •
      •
```

Function Subprogram

```
      FUNCTION FINE(L,X,YO)
      •
      •
      M = L+J/K
      •
      •
      FINE = X**M/YO
      RETURN
      END
```

In this example, the values of N,P and R are transferred to the subprogram arguments L, X and YO respectively. The value of FINE is computed, and this value is returned to the calling program where the value of RESULT is computed.

If a dummy argument is an array name, an appropriate DIMENSION or specification statement must be written in the Function subprogram. None of the dummy arguments can be listed in EQUIVALENCE, COMMON or DATA statements in the subprogram.

Calling program

```
XRAY = 11.1
RESULT = XRAY + 2.3 + ALPHA(2.6, 0.5, XRAY)
```

Function subprogram

```
FUNCTION ALPHA (BETA, GAMMA, DELTA)
X = BETA**2 - GAMMA**3
DELTA = X**2.5
ALPHA = DELTA**2
RETURN
END
```

SUBROUTINES

A subroutine is similar to a Function subprogram in that it is an independent program unit in subprogram form, but unlike the Function, a subroutine has no value associated with its name, therefore it cannot be executed simply by writing its name; a CALL statement must be given to bring the subroutine into operation.

The subroutine name may consist of from one to six alphanumeric characters, the first of which must be a letter. Unlike variable and function names the first letter is not significant as there is no mode associated with the subroutine name.

If a main program needs to use a subroutine, control is transferred to that subroutine by the statement

```
        CALL S (argument 1, argument 2, .., argument n)
or      CALL S
```

S represents the symbolic name of the subroutine.

The actual arguments may be:
- Hollerith constants
- variable names
- array or array element names
- another subprogram name
- any other expression.

and they must agree in order, number and type with the corresponding dummy arguments in the subroutine. When a Hollerith constant is used as an actual argument, the address of the first character is transferred to the corresponding dummy argument (irrespective of the type of the dummy argument).
A subroutine must begin with the statement

```
        SUBROUTINE X (a1,a2,..,an)
or      SUBROUTINE X
```

X is the subroutine name (which must not appear in any other statement in the routine), the a's are dummy arguments; these must not be listed in any EQUIVALENCE, COMMON or DATA initialization statement that occurs in the subroutine.

A subroutine may contain any statement other than FUNCTION, BLOCK DATA or another SUBROUTINE statement.

A subroutine can use any of its arguments to return values to the calling program. Any arguments used like this must appear on the left-hand side of an arithmetic assignment statement or in an input list within the subroutine.

A subprogram may have arrays of adjustable size; the DIMENSION statement in the subprogram contains integer variables instead of integer constants. These variables must be written in the dummy argument list and given values by the calling program. The following example illustrates the use of adjustable dimensions. The subroutine can be used to fine the largest element in a particular row of any square array.

The CALL statement in the main program could be:

```
      CALL LARGE (ALPHA, 50, 5, BIGGST, M)
```

and the subroutine:

```
      SUBROUTINE LARGE (ARRAY, N, I, BIG, J)
      DIMENSION ARRAY (N,N)
      BIG = ABS(ARRAY(I,1))
      J= 1
      DO 15 K = 2,N
      IF (ABS(ARRAY(I,K)) .LT. BIG) GO TO 15
      BIG = ABS (ARRAY(I,K))
      J = K
   15 CONTINUE
      RETURN
      END
```

With the actual values 50 and 5 passed to the dummy arguments N and I, the subroutine will find the largest element in row 5 of a 50x50 array named ALPHA, placing the largest element in BIGGST and its row number in M.

In the next example, the main program reads a number N, and then N numbers into array A, and calls the SØRT routine to sort the array elements into ascending order. The subroutine compares adjacent numbers and exchanges their positions if necessary.

Main program

```
      DIMENSIØN A(100)
1     FØRMAT (I3/(6F12.4))
2     FØRMAT (10F12.4)
      READ (2,1)N, (A(I), I = 1,N)
      CALL SØRT (A,N)
      WRITE (3,2) (A(I), I = 1,N)
      STØP
      END
```

Subroutine

```
      SUBRØUTINE SØRT (A,N)
      DIMENSIØN A(N)
      K = N-1
      DØ 10 I = 1,K
      M = N-I
      DØ 10 J = 1,M
      IF (A(J+1) - A(J)) 5,10,10
5     TEMP = A(J)
      A(J) = A(J+1)
      A(J+1) = TEMP
10    CØNTINUE
      RETURN
      END
```

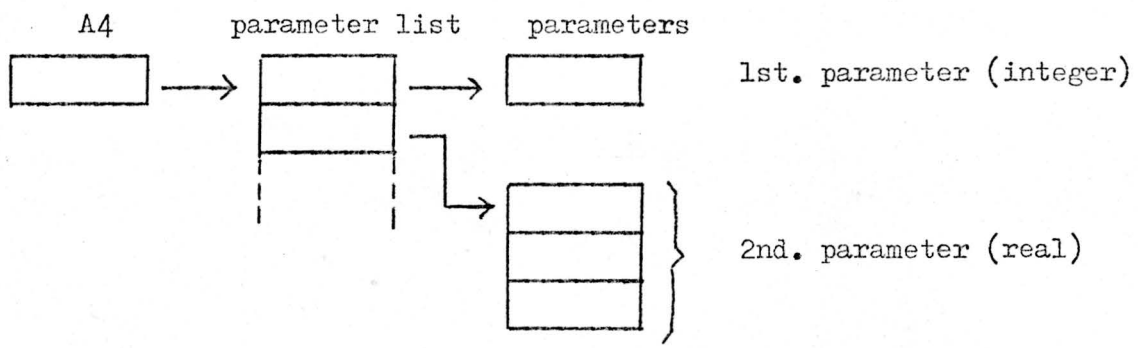
ASSEMBLY LANGUAGE SUBPROGRAMS

Subroutine or Function subprograms may be written in Assembly language, and the resultant object modules link-edited or link-loaded with FORTRAN modules.

The Assembly language subprogram must contain an entry point whose name is the subprogram name.
At execution time, the subprogram call is interpreted as a

CF A14, X

instruction, branching to the entry point, X.
When this instruction is executed, the A4 register contains the address of the first word of the parameter list which itself contains the address of the first actual parameter; the second word contains the address of the second parameter, and so on.



When execution of the subprogram is completed, control is returned to the calling program by the instruction

RTN A14

The subprogram cannot use the stack (which is defined by the calling program) or A14 (which is the stack pointer, indicating the first free location on the stack) in any way other than as shown.

If a CF instruction is used in the subprogram, the stack must then be defined by the programmer and the contents of A14 saved.

If the subprogram is a Function, returning a computed value to the main program, an integer or logical value will be returned to A1, a real value to A1 - A3 (mantissa in A1, A2; exponent in A3), a double precision value to A1 - A4, and a complex value to A1 -A6 (real part in A1 - A3, imaginary part in A4 - A6).

BLOCK DATA SUBPROGRAMS

A separate subprogram must be written to initialize variable values in a labelled common block. This subprogram contains only DATA, COMMON, EQUIVALENCE and DIMENSION statements, plus any Type statements relative to the data being defined.

The form of the statement is

```
BLOCK DATA
.
.
.
END
```

The BLOCK DATA statement must be the first statement in the subprogram.

Values cannot be entered into elements of blank common blocks.

All elements of a common block must be listed although they need not all be initialized.

Example

```
      BLOCK DATA
      COMMON/BLOK/R,S,T/RUN/X,Y,Z(2)
      DIMENSION T (5)
      COMPLEX R
      INTEGER Z(2)
      LOGICAL X,Y
      DATA R/(0.71,2.96)/, (T(I),I=1,5)/2.5,17.0,2*0.55,8.3/,
     1     X/.FALSE./,Y/.TRUE./,Z(1),Z(2)/4,8/
      END
```

Input/Output statements are of two types;

- READ and WRITE statements
- Auxiliary I/O statements

READ and WRITE statements cause data records of sequential files to be transferred to and from internal memory.
Auxiliary I/O statements consist of BACKSPACE and REWIND which are used for positioning of an external file, and ENDFILE which indicates divisions on an external file.

READ and WRITE STATEMENTS

READ and WRITE statements specify transfer of information in the form of formatted or unformatted records.
A formatted record consists of a string of ASCII characters. The transfer of such a record requires that the READ or WRITE statement contains a reference to a format specification which supplies the necessary positioning and conversion specifications.
An unformatted record consists of a string of values in binary code.
The statements have the following forms:

formatted:	READ(u,f)k	or	READ(u,f)
	WRITE (u,f)k	or	WRITE(u,f)
unformatted:	READ(u)k	or	READ(u)
	WRITE(u)k		
direct			
formatted:	READ(u's,f)k	or	READ(u's,f)
	WRITE(u's,f)k	or	WRITE(u's,f)
direct			
unformatted:	READ(u's)k	or	READ(u's)
	WRITE(u's)k		

In each case, u is a file code (an integer constant or non-subscripted integer variable whose value has been defined) which is known to the Monitor, and which designates an input/output device; f (in formatted statements) is the label of a FORMAT statement; k is a list of variables, array names and array elements; and in direct access, s is an integer constant or variable indicating the number of a sector in file u. When s is an integer variable, its value is not modified on completion of a READ or WRITE statement. When direct unformatted READ or WRITE are used, both logical and physical records have a fixed length of 200 words.

The associated statement whose label (f) appears in a READ or WRITE statement is the FORMAT statement. This indicates to the computer the form of the numbers being read or written, i.e. the I/O statement specifies what is to be read or written, the FORMAT statement specifies how and where the information is placed within a record.

I/O LISTS

The list (k) of an input/output statement may be of any length, and names the variables in the order they are to be transmitted.
The list elements may be variable names, array names, array elements and implied DO's. On input, these elements receive the values of the data being read. When an input statement is executed, the values read are assigned to the corresponding list elements, i.e. the first value read is assigned to the first element in the list, and so on. In this way, an integer value read early in the list may be used as a subscript elsewhere in the list. It must, however, appear as an input variable before it is used as a subscript.

Example

```
READ(u,f)N, A(N), I, J, XARRAY(J), Y
```

When variables appear in an input list with variable subscripts, these subscripts need not necessarily be defined within the same list (as they are in the previous example); their subscript values may have been defined by other statements - in a DO-loop, for instance.
The example following shows a READ statement which will be executed ten times. Each time it is executed a value will be read and placed in the A array, its position determined by the value of the DO-index.

```
4   DØ 5 I=1,10
5   READ (01,7) A(I)
```

To transfer a complete array, only the array name need be written, without the subscripting information. All the array elements will then be given values. (The array must be declared elsewhere in the program, in a DIMENSION, COMMON or Type statement specifying the array dimensions).

IMPLIED DO'S

Variables within an I/O list may be indexed and incremented in the same way as those in a DO statement. The variables and their indexes must be enclosed in parentheses.
For example, the statement

```
READ (u,f) (A(I),I=1,50)
```

is equivalent to the statement

```
READ (u,f) A(1),A(2),A(3),....,A(49),A(50)
```

Although only executed once, the self-indexing statement will cause 50 numbers to be read into successive elements of the array A.

Implied DO's may also be nested, for example, given an array of 2 by 3 by 5 elements, the statement

```
READ (u,f) (((ARRAY(I,J,K),K=1,2),J=1,2),I=1,1)
```

will cause data to be assigned in the following order:

```
ARRAY(1,1,1); ARRAY(1,1,2); ARRAY(1,2,1); ARRAY(1,2,2)
```

As a further example:

```
READ (u,f)I,(C(J),J=1,I)
```

The variable I is read first and its value then used as an index to specify the number of values to be read into the array C.

FORMATTED AND UNFORMATTED RECORDS

A formatted record consists of a string of ASCII characters. The transfer of such a record requires a FORMAT statement to specify - on output - the layout of the external record, the form in which the data must be written, and - on input - the layout and how the data being read are to be interpreted. Within a format specification records are defined by means of a slash (/) separator. This means that FORTRAN logical and physical records, read or written via Monitor I/O requests, are identical. The maximum record length allowed in formatted READ or WRITE depends upon the I/O device associated with the specified file code.

An unformatted record is a direct binary representation. Data are transferred between the I/O buffer and the FORTRAN program or the output device without any conversion, except where the output unit is a line printer or the operator's typewriter; in this case an unformatted WRITE is interpreted as a hexadecimal core dump. (Unformatted input from the operator's typewriter is not possible).
The length of a physical record on unformatted files is variable, its maximum value depends on the device. The first two characters of any record identify the record as unformatted.

In unformatted I/O, the logical length is determined by the number and types of the variables in an I/O list. If the length of a logical record is less than (or the same as) the maximum physical record length, only that one logical record will be recorded on the physical record. If the length of a logical record is greater than the maximum physical record length, the logical record will be continued over as many consecutive physical records as is necessary.

When direct access unformatted READ or WRITE are used, both logical and physical records have a fixed length of 200 words.

Formatted READ

READ (u,f)k or READ (u,f)

The next physical records are read from an external file. Data are input from the device specified by file code u and converted according to the format specification f; the values are assigned to the list elements (if any). If no variable list is specified, one record will be read but ignored, resulting in a forward skip.

When a READ statement is executed under format control, one record is read when the READ process is started; succeeding records will be read when slashes are found during the scanning of the format specification. When input is from the operator's typewriter, a request for a new record is made by the bell ringing and a question mark being typed out.

If the format control reaches the outermost right parenthesis of the format specification and there are list elements still to be input, scanning is resumed from the beginning of the last group repeat specification. If there is no group repeat specification, control reverts to the beginning of the format specification , and scanning continues from left to right.

If more values are required for the list than are in one record, and the format control does not initiate reading of another record, i.e. there is no slash in the format specification, an error message is output and processing stops.

Formatted WRITE

WRITE (u,f)k or WRITE (u,f)

Succeeding records are written from memory to the specified medium. The list elements are converted according to the format specification and output to file u.
A new record is written when a slash is found or when format control ends. If more information is offered by the list than can be contained in one record, and the format control does not initiate the output of another record, an error message is generated and processing stops.

If no list is specified a blank record is inserted in the output stream.

Unformatted READ

READ (u)k or READ(u)

One logical binary record will be read from the file identified by u, and the values assigned to the corresponding list elements, if any. If the number of values input from the record does not correspond to the number required by the list, an error message will be output. If the list is omitted, a record is skipped.

Unformatted READ from the operator's typewriter is not possible.

Unformatted WRITE

WRITE (u)k

The values specified by the list k are output to the specified file in binary format. As the data are in binary form, no format specification is necessary.
Output to the typewriter or line printer is represented by four hexadecimal characters for each word output.

Direct Access

Direct Formatted READ

READ (u's,f)k or READ (u's,f)

Execution of this statement causes the specified sector (s) of file u to be read.

Direct Formatted WRITE

WRITE (u's,f)k or WRITE (u's,f)

This statement causes data to be written on the specified sector of file u.

Direct Unformatted READ

READ (u's)k or READ (u's)

This statement causes one logical to be read from the sectors on file u. If the numbers of values required by the list k exceeds the number on the sector, the next sector of the file will be read until k is satisfied.

Direct Unformatted WRITE

WRITE (u's)k

The values specified by the list k are output to the indicated sector of file u; if the values of k exceed 200 words, the remaining values will be output to the sector following.

AUXILIARY I/O STATEMENTS

ENDFILE statement

ENDFILE u

Execution of this statement causes and End-Of-File to be written on the specified file, and indicates a demarcation of a sequential file.

REWIND statement

REWIND u

This statement is used only for magnetic tape units and sequential disc files, and causes the file to be positioned at its initial point.

BACKSPACE statement

BACKSPACE u

This statement applies to magnetic tape units only; the file is backspaced by one record , unless already at its initial point in which case the instruction will have no effect.

IOSTAT FUNCTION

A library function called IOSTAT is available which tests the status returned by the Monitor after each READ or WRITE. (For details on the status word, see P855/P860 Disc Monitors). This function allows the programmer to test an End-Of-File, or some other special condition.

If bit 0 of the status word is set to 1, an error message is output and the job aborted.

If the status word has a positive, non-zero value, the current READ or WRITE statement is abandoned (even if the whole of the I/O list has not been processed) and control is passed to the next statement in sequence in the FORTRAN program.

If the status word is 0 the I/O statement is processed as normal.

IOSTAT is an integer function with one integer argument. (This argument is ignored but must be coded to prevent a syntax error as FORTRAN does not accept functions without arguments). The value of the function IOSTAT is the last status returned by the Monitor for a READ or WRITE operation.

For example,

```
READ (1,10) (A(I),I=J,K)
IF (IOSTAT (0) .EQ. 1) GO TO 200
```

will cause data to be read from file 1, and then a branch made to the statement with the label 200 when an End-Of-File has been read.

Standard File Codes

(For Basic Executive Monitor)		(For Disc Operating System)
01	Source input	01 Operator's typewriter
02	Listing output	02 Print unit
03	Punch output	03 Punch unit
04	Operator's typewriter(input and output)	
06	ASR tape reader	10 to 207 = For user assignment
07	ASR tape punch	others- Reserved
08	Paper tape reader	
09	Paper tape punch	
10	Reserved for disc units	
11		
12		
13		

Format statements are non-executable statements which are used in conjunction with the input and output of formatted records to provide conversion information between the internal representation and the external character strings. All FORMAT statements must be labelled so that they can be referred to by READ or WRITE statements, and they may be placed anywhere in the program.

The general form of the statement is:

FORMAT (format specification)

The specification is written

FORMAT (q_1 t_1 z_1 t_2 $z_2 \ldots t_n$ z_n q_2)

where q is a series of slashes, or is empty;
t is a field descriptor or group of field descriptors;
z is a field separator.

FIELD SEPARATORS

A field separator may be a slash (or series of slashes) and/or a comma. (A slash indicates demarcation of formatted records).

A slash indicates either the beginning of a new record or the end of the preceding one. If there are no slashes within a FORMAT statement, a FORTRAN record is defined as being from the beginning of the format specification (left parenthesis) to the end (right parenthesis). A new record is written when the format control ends at the outer right parenthesis.

Example

```
xxxxx FORMAT (..., ..., ...)
              <-------------->
              1 FORTRAN record
```

If slashes occur within a FORMAT statement, a record is defined as being from the beginning of the format specification to the first slash, or from one slash to the next, or from the last slash to the end of the format specification.

Example

```
     xxxxx FORMAT (..../..../....)
                   <--> <--> <-->
                    |    |    |
                    L----+----+---- each corresponds to one record.
```

Blank output may be introduced or input records may be skipped by using consecutive slashes.
If there are n consecutive slashes at the beginning or end of a FORMAT statement, n input records are skipped or n blank records are inserted between output records - each one output is represented by a single blank character. If n consecutive slashes appear anywhere else in the statement, the number of records skipped or blanks inserted is n-1.

REPEAT SPECIFICATION

This is an integer constant preceding a field descriptor, or a group of field descriptors, and indicates the number of times that particular conversion is to be repeated.

Example

(2F7.3, 4I5) is the same as
(F7.3, F7.3, I5, I5, I5, I5)

If a group of field descriptors is to be repeated, the group is enclosed within parentheses and preceded by the desired repeat count, indicating the number of times to interpret the enclosed grouping.

Example

(F8.5, I2, 3(E7.4, F5.2))

This will cause each of the descriptors within the inner set of parentheses to be repeated three times. These descriptors will be repeated alternately, i.e. E7.4 then F5.2 then E7.4 again, and so on. This is quite different from

(3E7.4, 3F5.2)

which specifies three E7.4 fields followed by three F5.2 fields.

In addition to the parentheses required by the FORMAT statement, nine levels of group parentheses are allowed.

Example

(I2, 2(E9.4), 3(I5, E6.2))

FIELD DESCRIPTORS

Field descriptors specify the type and form of conversion between internal representation and external notation.
The field descriptors have the forms:

rIw
srFw.d
srEw.d
srGw.d
srDw.d
rLw
rAw
nX
nH$h_1h_2h_3....h_n$

where s is optional and represents a scale factor;
r, the repeat specification, is also optional and represents a positive, unsigned integer constant indicating the number of times the field descriptor it precedes is to be repeated;
w and n are positive, unsigned integer constants representing the field width of the external character string (including any plus or minus signs and a decimal point);
d is also unsigned, an integer constant and represents the number of positions in the field occupied by the fractional part of the external character string;
the letters I, F, E, G, D, L, A, X and H indicate the type of conversion to be used;
each h is one ASCII character. In place of the nH descriptor, Hollerith output data may be represented by a string of characters enclosed within apostrophes.

The format specification must be enclosed in parentheses.
The values of r, n and w must be greater than zero.

SCALE FACTOR

The internal or external data representation may be modified by the use of a scale factor. The effect of the scale factor is that the external number is equal to the internal number times ten raised to the power of the scale factor.
The field specification is:

nP or -nP

where n is an integer constant. This specification precedes a format descriptor.

A scale factor may be used with the F, E, G and D descriptors, provided none of these contain a decimal exponent in the input string. Whenever a scale factor is written with a field descriptor, it applies to all succeeding F, E, G and D descriptors until another scale factor is established. If a given scale factor is to be applied to one field descriptor only, the next descriptor must have an OP scale factor.

Input

A scale factor is ignored if any of the specifications contains a decimal exponent, otherwise it is defined as

external quantity = internal quantity * 10^n

Examples

Field descriptor	Input characters	Internal value
-3PE12.4	bbbbbb123456	+12345.6
-2PE12.4	-bbbbb123456	-1234.56
2PE12.4	bbb-bb123456	-b.123456
1PF11.4	bb-175.8041	-17.5804

Output

The E, D or G output specifications may contain an E or D decimal exponent. A positive scale factor used with a field descriptor containing an exponent multiplies the number by 10^n and decreases the exponent by n.
F output has the same format as F input; for G output the scale factor has no effect if the value to be converted is within the range that allows F-type conversion to be effectively used.

Examples

Field descriptor	Internal value	Characters output
1PE13.3	55.3647	bbbb5.536Eb01
-3PE12.4	+123456	bbb.0001E+09
-1PD12.4	+123456	bbb.0123D+07
-1PF11.4	-175.8041	bbb-17.5804
1PF8.2	-123.456	-1234.56
2PG12.4	0.012345	bb12.345E-03
2PG12.4	-1234.56	bb-1235.bbbb

Format Specification in READ and WRITE Statements

The FORMAT statement reference in a READ or WRITE statement may be specified by a label, an array name or a Hollerith constant which constitutes a valid FORMAT specification. When the format specification is in an array, an nH descriptor may form part of the specification.

Example

```
      WRITE (2,'(6X, "A=", F15.4)')A
    is equivalent to
      WRITE (2,333)A
  333 FORMAT (6X, 'A=', F15.4)
```

In the first WRITE, quote marks inside the Hollerith string must be coded as double apostrophes.

I-type descriptor (Integer)

The field specification is Iw (or rIw)
w defines the number of characters to be processed, including sign (where applicable) and any blanks.

On input w characters are converted to a 16-bit integer. Leading blanks or spaces are not significant; others within the field width are interpreted as zeros. The range of any integer value, i, is

$$-32767 \leqslant i \leqslant +32767$$

Examples

Field descriptor	Characters input	Internal value
I3	b+1	+1
I5	bbb-5	-5
I7	bb23451	23451

On output the number is right justified in the field and is preceded by a minus sign if negative, or a blank if the number is positive. If the field width is not large enough to contain the stored number, the whole of the external field is filled with asterisks.

Examples

Field descriptor	Internal value	Characters output
I5	23456	23456
I3	-2	b-2
I2	125	**
I5	-12345	*****
I6	-225	bb-225
I3	0	bb0

F-type descriptor (Real)

The field specification is Fw.d (or srFw.d)
w defines the number of characters to be processed; d defines the fractional part.

On input the external character string consists of an optionally signed digit string which may contain a decimal point. If the number read does contain a decimal point, the d part of the specification will be ignored. If no decimal point is present, the number will be converted according to the specification.

The basic form of the external character string is as follows:

- leading blanks (these are ignored)
- a + or - sign (an unsigned input is assumed to be positive)
- an integer part
- a decimal point
- a fraction part

Each of these elements is optional and this basic form may be followed by an exponent which has the form

- letter E
- an optional + or a - sign
- a decimal exponent in the form of an integer constant, consisting of not more than three digits.

Examples

Field descriptor	Characters input	Internal value	
F6.2	b12344	123.44	
F8.4	bbb35711	3.5711	
F5.2	2.255	2.255	the input number contains a decimal point, so the d specification is ignored.

On output the external basic form is used, without an exponent. The number is right justified in the field, preceded by blanks where necessary and a minus sign if negative. If the fraction is larger than the d specification requires, it will be rounded. (This rounding -off may cause a very small number to appear to be zero). If d is larger than the fraction, zeros will be added on the right hand side. If the w specification is too small, the whole of the external field is filled with asterisks.

Examples

A format specification of F5.2 specifies a total field width of five positions (including one position for the decimal point), and would have the following output formats:

Internal value	Characters output	
25.37	25.37	
505.25	*****	(w specification too small)
81.5	81.50	
2.125	b2.12	
-9.	-9.00	
-.6	-b.60	
6.9821	b6.98	(last two digits of accuary lost)
5.757	b5.76	(fraction rounded)

E-type descriptor (Real exponent)

The field specification is Ew.d (or srEw.d)

The number of characters specified by w is converted to a floating point number with d digits to the right of the decimal point.

On input the format is the same as F input; the E exponent may be included or omitted (as with F-type).
The exponent is the power of ten by which the number must be multiplied to obtain its true value.

Examples

Field descriptor	Characters input	Internal value
E10.3	-0.200E-02	-.002
E10.5	+27.567E-3	.027567
E10.5	+27.567E+3	27567.0
E12.4	bbbbbbbbbbb5	0.0005
E12.4	-12345678E-3	-1.2345678
E10.5	567891E-01	.567891

On output the number is right justified, the field is preceded by blanks where necessary, and a minus sign if the number is negative; this is followed by a decimal point, a decimal fraction part of d positions, and a decimal exponent of the form:

$$E\ y_1y_2y_3$$

where each y is a decimal digit.

If the exponent cannot be contained in three positions, the whole external field is filled with asterisks.

Examples

Field descriptor	Internal value	Characters output
E12.5	222.55	bb.22255E+03
E10.3	.0000000009	bb.900E-09
E10.3	535.	bb.535E+03
E14.5	276.55	bbbb.27655E+03
E10.3	-.002	-b.200E-02
E11.4	+123456	bb.1235E+06

G-type descriptor

The field specification is Gw.d (or srGw.d).

The external field occupies w postions, with d significant fractional digits.

On input the external format of the numbers to be input is the same as F-type. In memory the number is represented as a single precision floating point number.

Examples

Field descriptor	Characters input	Internal value
G12.7	bbbbb-123456	-.0123456
G12.0	bbbbb+123456	+123456.
G9.3	bbb175463	175.463

On output the form depends on the magnitude of the internal floating point number. If the decimal exponent of the internal number is greater than d or less than zero, an E-type conversion will be used; otherwise if the exponent is less than or equal to d, and greater than or equal to zero, an F-type conversion will be used, but modified as follows:

F(w-4).(d-e),4X

The decimal point must immediately precede, follow, or be incorporated in the d string of digits; the external field consists of w-4 positions, and four blanks are added to the value.

Examples

Field descriptor	Internal value	Characters output
G12.4	136.7632	bbb136.7bbbb
G12.5	82.43441	bb82.434bbbb
G14.6	.12345123	bbb.123451bbbb

D-type descriptor (Double Precision)

The field specification is Dw.d (or srDw.d).

The external field consists of w positions, the fractional part of which is d digits. The value is internally represented in double precision floating point format (3 word mantissa and 1 word exponant).

On input the external field is the same as an F-type format.

Examples

Field descriptor	Characters input	Internal value
D12.4	bbbbbbbbbbb5	0.0005
D12.4	bbbbbbbbbb5.	5.0
D10.3	bbb7234bb1	7234.001

On output the format is the same as for the E-type descriptor except that the letter D replaces the letter E in the exponent part.

Examples

Field descriptor	Internal value	Characters output
D10.3	238.	bb.238D+03
D11.4	654321.	bb.6543D+06
D10.3	.00000000003	bb.300D-10
D15.8	12.34567890	bb.12345679D+02

L-type descriptor (Logical)

The field specification is Lw (or rLw)

On input the first T or F in the next w characters of the input string causes a value of .TRUE. or .FALSE. to be assigned to the corresponding logical variable. Internally, a logical value is represented as an integer, the value of which is -1 for True and 0 for False.

On output the letter T or F (corresponding to the value of the logical variable) is inserted in the output record, preceded by w-1 blanks.

A-type descriptor

In the Aw specification w alphanumeric (ASCII) characters are read into or written from the specified list element. As 8-bit code is used for each character, each word of memory holds two characters. The associated variable can be of any type, so an integer or logical variable contains 2 characters, a real variable 6 characters, a double precision variable 8 characters, and a complex variable 12 characters.

The Aw descriptor causes w ASCII characters to be read into, or written from, w consecutive character locations from the address of the first character of the specified list element. If w is greater than the number of characters that can be held by the variable specified, successive variables are read until the w characters are transferred. If w is less than the "character capacity" of the variable, w characters will be input followed by trailing blanks, and for output, the w characters will be right justified, and preceded by as many blanks as is necessary to complete the total field width.

X-type descriptor

The field specification is nX.

This descriptor governs the insertion of blanks.

On input the next n characters are skipped.

On output n blanks are inserted in the output record.

H-type descriptor (Hollerith)

Hollerith (alphanumeric) data may be transferred by means of the nH field descriptor.

Hollerith information is read into or written from the n characters (blanks included) following the letter H in the format specification list.

On input the n characters in the format are replaced by n characters from the input field.

On output the n characters following the format code are written as part of the output record.

For output only, this field descriptor may be replaced by an ASCII character string enclosed in apostrophes. The data are then transferred exactly as written. When an apostrophe is used as part of such a string, it must be coded as a double apostrophe.

A 0H descriptor is not accepted (illegal format specification is diagnosed).

Printing of Formatted Records

When formatted records are printed, the first character of the record (which is not output) determines the vertical spacing:

0 indicates double line spacing
1 " skip to first line of next page and single spacing
+ " no advance
any other character indicates single spacing.

For each I,F,E,G,D,L or A descriptor in a format specification, there is a corresponding element specified in the input/output list. However, there is no corresponding list element for H or X descriptors, and the format control communicates information directly with the record.

The Full FORTRAN IV compiler runs under control of any Monitor (but if disc storage is used, can only run under either of the two Disc Monitors)and accepts both Basic and Full FORTRAN source programs.

The minimum configuration required is a P855 or P860 CPU with 12k words of memory and the ASR typewriter. However, source programs may be input from any of the input perpherals that can be attached to a 12K configuration, and system programs may be stored on disc and magnetic tape units.

The FORTRAN system consists of a compiler and a system library which contains I/O functions and Mathematical and Arithmetic routines. Both the compiler and the system library may be permanently stored on disc. (Details of disc operation and access can be found in the P855/P860 Disc Monitors manual.)

The compiler makes one pass over the FORTRAN source modules (main program and any subprograms called by the main program) and produces corresponding object modules, ready for processing by the Link Editor/Loader. Each source module may be compiled invidually. The input to the Link Editor/Loader is:

1) the FORTRAN main program,
2) all external subprograms written by the user,
3) the FORTRAN System Library.
   (The input order of the first two is unimportant, but the Library modules must be the last to be input.)

The Link Loader selects from the Library all the routines called by the program modules, and all other functions required for program execution. It then combines the various modules into one relocatable object program which is ready for execution.

Error messages that are found during compilation are printed on a new line immediately after the source line in which the error was detected (not necessarily after the line in which the error first occurred).The error message has the form:

F/MESSAGE

followed by a hexadecimal error code (a list of the codes and their meanings is given at the end of the chapter).

COMPILER CONTROL STATEMENTS

Two special statements are available which control the compilation process; these are IDENT and OPTIONS.

IDENT Statement

The IDENT statement must be used to assign a name to any source (and its corresponding object) module. The statement is written on the first line of the module in the form:

IDENT␣m

where m is an identifier

This identifier has no connection with the FORTRAN program itself, but is used during the link-edit or link-load process to identify the object module. Where the object module is a subprogram, the identifier may be the same as, or different from, the subprogram name. Any Block Data subprogram must be preceded by an IDENT statement.

OPTIONS Statement

The OPTIONS statement may be used to suppress listing, to control the conditional compilation feature, and to request the generation of a re-entrant module. This statement may be written anywhere in a source program, and has the form:

OPTIONS i

where i is an option or list of options separated by commas.

The options are specified by the letters L,X,D or M which are associated with four corresponding boolean variables in the compiler's work area. When an IDENT statement at the beginning of a module is processed, these variables are initialized to zero. When the OPTIONS statement is processed the values of those boolean variables whose names appear in the OPTIONS list are reversed.

Example

Source program.

```
   IDENT PROG                      all booleans set to 0
   OPTIONS L                       value reversed so statement listing is
                                                              suppressed
    .
    .
X  A = B                           X boolean is 0, therefore this line is
    .                                                       not compiled
    .
   OPTIONS X                       value reversed
X  IF (M+(I-J** 2)) 10,15,15       this line will be compiled (boolean =1)
    .
    .
```

L (Suppress Listing)

When the L variable is set to 1 there will be no source listing. Error messages, however, will still be listed.

X (Conditional Compilation)

When the boolean variable X is 0 any source line whose first character is the letter X is ignored by the compiler.

When the variable is set to 1, the source line will be compiled, the X will be ignored and processing will start from the second character in the line.

D (Generation of a Re-entrant Module - Dynamic Allocation)

When the boolean variable D is set to 0, the output object module will not be re-entrant.

When the variable is set to 1, the work storage for the output object module is obtained at run time by means of a dynamic memory allocation request to the Monitor. This object module will then be re-entrant, i.e. it can run concurrently at different software levels and can be interrupted at any stage at a request from another program of a higher priority.
Care must be taken not to modify the format descriptors at run time. The D option is not accepted if the module contains a DATA statement.

FORTRAN JOB CONTROL

When the FORTRAN compiler is stored on disc, a job control command must be given to call the compiler and the source modules into memory. Under the disc system, source programs must be loaded onto disc from external input peripherals before being compiled. All source programs are stored on the temporary disc file /S. The Read Source command which copies a source module onto the temporary disc file has the form:

RDS␣⌈/<file code>⌉

If the user wishes to make the file permanent the RDS command may be followed by a Keep File command which stores the program in the user's library. This is written:

KPF␣/S⌈,<name>⌉

<name>is the program identifier; if it is not specified the name in the IDENT control statement is taken.

The FORTRAN job control command is:

FOR␣⌊/S|<name>⌋ ⌈,NL⌉

This command must be used to call the compiler into memory and to compile a FORTRAN source program from the temporary file /S or from the user's library.

One of the items within the ⌊ ⌋ brackets must be specified. /S indicates that the source program is to be compiled from this file; <name> is the program identifier and indicates that the source program is in the user's library.

If NL is specified, no listing of the compiled program will be provided. Where a listing is given it is output to the print unit. Error messages output on the operator's typewriter may be 'I/O ERROR' (where there is an irrecoverable I/O error) and 'COMPILATION NOT SUCCESSFUL. NO OBJECT CODE PRODUCED'. All other error messages are output to the print unit (with listing).

The object module is output to the temporary object code file /O. If the user wishes to have an object module on paper tape, the POB control command can be given, and by means of the KPF command the object module can be transferred to the user's library.

Error Codes

Code	Meaning
0000	Invalid statement number
0001	'TO' missing in an ASSIGN statement
0002	Variable in an ASSIGN statement is not integer
0003	Invalid character ending an ASSIGN statement
0004	FORMAT label instead of a statement label in an ASSIGN statement
0100	Left part of an assignment statement invalid
0101	Invalid character ending an assignment statement
0102	Formal argument name given as a statement function name
0103	More than 32 statement functions
0104	More than 8 arguments in a statement function
0105	A statement function's name is used as one of its arguments
0106	Right parenthesis following last argument of a statement function definition is missing
0107	The = sign in a statement function definition is missing
0108	Invalid character ending a statement function definition
0109	Statement function definition not allowed in Block Data subprogram
0200	CALL must be followed by a subroutine name
0201	Name following CALL is already specified as other than a subroutine name
0202	Name following CALL is already specified as a FUNCTION name
0203	Invalid character ending a CALL statement
0300	The first character of a FORTRAN statement must be a letter
0301	Unclassified statement
0600	Illegal variable or array name in DATA
0601	Illegal delimiter in DATA
0602	Formal argument in DATA
0603	Illegal variable or array type inside BLOCK DATA
0604	Common variable outside BLOCK DATA
0605	Illegal subscript expression in DATA
0606	Subscript overflow in DATA
0607	Illegal control variable in a DO-implied Loop
0608	Control variable in a DO-implied loop does not correspond to any subscript.
0609	Illegal parameter in a DO-implied loop
060A	Terminal value less than initial value in a DO-implied loop

060B Increment of zero in a DO-implied loop

060C Illegal variable list in DATA statement

060D Zero repeat factor in DATA constant list

060E Constant list does not correspond to variable list in DATA statement

060F Illegal constant in DATA statement

0701 Invalid character in a declaration statement

0800 Control variable not specified or not integer in a DO statement

0801 An = sign must follow the DO-loop control variable

0803 Illegal number of parameters in a DO statement

0804 Too many commas in a DO statement

0900 A DO statement may not end another DO statement

0901 A DO statement may not be the second part of a logical IF statement

0902 DO must be followed by a statement number

0903 The statement label specifying the end of the DO must not be a FORMAT label

0904 The end of DO-loop statement label not defined

0905 DO ends in an invalid character

0A00 An EQUIVALENCE group declaration must begin with a left parenthesis

0A01 Only actual array/variable may be specified in an EQUIVALENCE statement

0A02 Any subscript element must be an integer constant in an EQUIVALENCE statemen

0A03 EQUIVALENCE group declaration must end with right parenthesis

0B00 A name which is already specified in a DIMENSION/EXTERNAL/SUBROUTINE/FUNCTION statement appears in an EXTERNAL statemen

0B01 A name already specified in an EQUIVALENCE statement appears in an EXTERNAL statement

0B02 A name already specified in a COMMON statement appears in an EXTERNAL statement

0B03 Unexpected character in an EXTERNAL statement

0C01 Non-executable statement in a module other than Block Data

0C02 Undefined label(s)

0C03 Incomplete Do-loop

0C04 Dynamic allocation with DATA initialisation

0C05 Executable statement in Block Data

0C06 No DATA initialisation in Block Data

0C07 RETURN missing in subprogram

0E00 Unexpected character ending an auxiliary I/O statement

1000 Number of arguments specified in two uses of the same subprogram is not the same

1100 FORMAT label missing in a FORMAT statement
1101 A FORMAT statement ends a DO-loop
1102 FORMAT label already defined as label of another FORMAT or as a statement label
1103 FORMAT label already referenced as a statement label
1104 'FORMAT' must be followed by (
1105) is missing at end of FORMAT statement
1106 A FORMAT may not be the second part of a logical IF statement

1400 Integer variable or constant requested and not found

1500 A name was requested and not found

1600 Non-FORTRAN character

1900 A COMMON block name must not be used as a variable name
1901 A digit must follow the decimal point of a constant which has no integer part specified
1902 The character following the E or D exponent must be a + or - or a digit
1903 The exponent part must be less than 32767
1904 The real/imaginary part of a complex constant must not be either logical or integer
1905) must follow the imaginary part of a complex constant
1906 Relational operator, logical operator or logical constant is incorrectly written (terminal period missing or insufficient letters
1907 A relational operator/logical operator/logical constant name was not defined
1908 A Hollerith constant written nH.. implies n> 0
1909 " does not appear within quote marks
190A A complex constant is not correct: real number missing
190B Too many numerical constants in this program unit
190C Hexadecimal constant overflow or $ not followed by a digit or a letter from A to F
190D Real constant overflow
190E Too many digits in a real constant

1A00 DO-loop ends in a GO TO statement

1A01 Unexpected character following GO TO

1A02) missing in computed or assigned GO TO

1A03 Comma missing in computed or assigned GO TO

1A04 Integer variable name not found in computed or assigned GOTO

1A05 Name in computed or assigned GO TO is not a variable name

1A06 Variable in a computed or assigned GO TO is not integer

1A07 Unexpected character ending a GO TO statement

1A08 More than one statement label reference in a simple GO TO

1A09 Illegal character ending a GO TO

1F00 Parenthesis error: IF must be followed by a (with a corresponding)

1F01 Invalid IF expression; such an expression may not contain an =

1F02 An arithmetic IF may not be used to end a DO-loop

1F03 More or less three statement numbers specified in an arithmetic IF

1F04 Unexpected character ending an arithmetic IF statement

1F05 The control expression of a logical IF statement is neither integer nor logical

1F06 A logical IF statement may not follow another one

2000 Parenthesis error in an I/O statement

2001 Invalid FORMAT label reference in an I/O statement

2002 Unexpexted FORMAT reference (neither an array name nor a Hollerith constant)

2003 Format reference is a name but not an array name

2004 No list and no format specified in a WRITE statement

2005 Incorrect variable in an I/O list (not a name nor an array)

2006 Unexpected character in an I/O list

2007 Incomplete DO-implied loop

2008 Invalid array subscript in an I/O list

2300 Invalid statement number

2301 Executable statement not allowed in Block Data

2400 Statement number already defined (as FORMAT or other statement number)

2600 A FORMAT label may not be specified in a list of statement labels

2800 A FORMAT or a statement number may not be zero

2C00 Unexpected character ending an OPTIONS statement

3200 A PAUSE or STOP statement may not end a DO-loop

3500 A RETURN statement may not be used in a main program
3501 A RETURN statement may not end a DO-loop

3601 A RETURN statement may not end a DO-loop

3700 DATA statement has been incorrectly processed
3B00 The requirements of local (non-common) variables and arrays exceed 16384 16-bit words
3B01 Number of dimensions declared in a DIMENSION statement does not correspond with those specified in EQUIVALENCE
3B02 Overflow in a COMMON block (more than 16384 words)
3B03 Inconsistency in declaration of groups of Equivalenced names, making allocation impossible
3B04 An array name which is specified in an EQUIVALENCE statement must be declared in a DIMENSION statement
3B05 Two COMMON variables may not be related in EQUIVALENCE
3B06 EQUIVALENCE declaration extends COMMON block backwards

3C00 Invalid sequence for current statement

3E00 Invalid sequence for SUBROUTINE or FUNCTION statement
3E01 A FUNCTION declaration has no argument specified
3E02 Invalid argument in a SUBROUTINE or FUNCTION statement
3E03 Argument in a SUBROUTINE or FUNCTION statement is duplicated
3E04 Illegal delimeter in a SUBROUTINE or FUNCTION statement

3F00 Misplaced common block name in COMMON statement
3F01 Illegal variable dimension in array declaration
3F02 Number misplaced in declaration
3F03 Illegal declaration (general)
3F04 Illegal delimeter in declaration
3F05 Illegal common block name
3F06 Slash missing in COMMON statement
3F07 Illegal array name in array declaration
3F08 Illegal number of dimensions in array declaration
3F09 Formal argument in COMMON statement
3F0A Common element defined twice
3F0B Inconsistent variable or array type
3F0C Dimension overflow in array declaration

4200 Operator incorrect in an arithmetic expression

4201 Hollerith constant in an arithmetic expression

4202 Incorrect character in an arithmetic expression

4203 Erroneous Function call

4204 No argument referenced in an intrinsic function call

4205 Error in arithmetic expression

4206 Hollerith constant in an arithmetic expression or impermissible type mixing

4207 Incorrect type in a logical expression

4208 Error in MIN or MAX function

4209 Subscript is not integer or there are more than 3 subscripts in an array element reference

420A Subroutine reference in an arithmetic expression

420B Argument type or number of arguments incorrect in an intrinsic function reference

420C Wrong number of arguments in a statement function

Run-Time Errors

01 No more core storage can be allocated for a re-entrant module, or for an I/O operation

02 Wrongly generated object code - ask software maintenance

03 Incorrect value given for index variable in a GO TO statement

04 Negative step value in a Do-loop

10 Overflow in integer arithmetic operation

11 Undefined result for ISIGN function

12 Overflow or undefined result in integer exponentiation

13 Arithmetic overflow in subscript computation or subscript not positive

20 Overflow in real addition or subtraction

21 Underflow in real addition or subtraction

22 Overflow in real multiplication or division

23 Underflow in real multiplication or division

24 Real division by zero

25 Overflow in real negation or in ABS or SIGN computation

26 Undefined result for SIGN function

27 Overflow in IFIX function

28 Undefined result in real exponentiation

29 Overflow in real exponentiation

2A Undefined result for ALOG function

2B Overflow in ALOG function

2C Negative SQRT argument

2D Overflow in EXP function

2E Undefined result for raising a real to an integer power

2F Overflow in raising a real to an integer power

30 Overflow in double precision addition or subtraction

31 Underflow in double precision addition or subtraction

32 Underflow in double precision negation

33 Undefined result for DSIGN function

34 Overflow in double precision multiplication

35 Underflow in double precision multiplication

36 Overflow in double precision division

37 Underflow in double precision division

38 Double precision division by zero

39 Negative argument to DSQRT

3A Undefined result for ALOG10 function

3B Overflow in DEXP function

3C Negative argument to DLOG function

3D Negative argument to DLOG10 function

3E Undefined result for ATAN2 function

3F Undefined result for DATAN2 function

40 Undefined result in double precision exponentiation

41 Overflow in double precision exponentiation

42 Undefined result when raising a double precision to an integer power

43 Overflow in raising a double precision to an integer power

44 Second argument to MOD function is zero.

45 Second argument to AMOD function is zero

46 Erroneous or un-normalized argument in real or double precision operation or function

47 Second argument to DMOD function is zero

48 Overflow in CONJG function

49 Overflow in DIM function

I/O Errors

70 Irrecoverable I/O error during an auxiliary I/O operation

71 Irrecoverable I/O error during a READ or WRITE operation, or illegal file code for a random I/O request

72 Illegal FORMAT specification

73 Field width too small or zero

74 Group or field repeat count is zero

75 Error in a string of ASCII characters between quotes in a format: Such a format is not allowed with a READ statement, or End of format statement encountered before last quotation mark.

76 No conversion specified in format for next I/O list element

77 Maximum number of characters allowed for a physical record on the specified unit is exceeded (I/O buffer overflow.)

78 First parenthesis of format specification is missing (when format reference is an array name)

79 Type of variable is not compatible with field descriptor

7A More than ten levels of parentheses in a format specification

7B Illegal logical variable (Input):
The first non-blank character is neither T nor F; or The whole external field is blank

80 Illegal input character (possibly a decimal point or exponent in an integer)

81 Overflow during input conversion.

82 Illegal unformatted record

83 Logical (unformatted) record is too small

84 Unformatted READ is not allowed on typewriter

85 Only one sector (400 characters) is allowed for an unformatted disc I/O.

CONTROL CHARACTERS IN SOURCE LINES

When source programs are input from the ASR 33 operator's typewriter or from paper tape, certain non-FORTRAN characters can be recognised by the compiler which facilitates the preparation of such source lines.

All source lines contain some non-FORTRAN characters which control the typesetting of a FORTRAN line. The input line is edited internally by the compiler and the result (the actual line to be compiled) stored in a buffer of 72 characters. The contents of the buffer are listed (unless the suppress listing option has been specified).

Control characters are used to mark the end of a line, to delete characters and to insert spaces. The control characters are:

LF	Line Feed
CR	Carriage Return
Xoff	Tape reader on ASR switched off
DE	Delete; this is ignored i.e. there is no corresponding character transmitted to the buffer. Characters are deleted by over-punching.
←	Horizontal arrow; indicates 'delete the preceding character'. The current postion or column in the buffer is decreased by one. Several ← characters in succession cause a corresponding number of characters to be deleted.
↑	Vertical arrow; indicates 'delete the whole of the current line' from column 1 up to and including the next CR.
\	Reverse slash is used for tabulation. Its interpretation depends on the number of the column n at the time it occurs, i.e.

if $n<6$: 6-n spaces are transmitted to the buffer (tabulate to column 6);
if $n=6$: no corresponding character is transmitted;
if $n>6$: 73-n spaces are transmitted (tabulate to end of buffer).

A maximum of 72 characters can be transmitted. If there are more than this they will be ignored; if there are fewer than 72, the line will be filled with spaces.

End of line on the operator's typewriter must be coded as

LF XOff CR DE

The sequence for any other device is

CR LF

Examples

Source line: \ ␣A=1.\ MØD10050 LF CR

FORTRAN line: ␣␣␣␣␣␣A=1.␣␣ ␣␣
1 6 72

Source line: \␣ SURB←←BROUTINE␣A(i) LF CR

FORTRAN line: ␣␣␣␣␣␣ SUBROUTINE␣A(i)␣ ␣␣
1 6 72

Source line: \ OB=↑ LF CR

500\ OB=3. LF CR

FORTRAN line: 500␣␣OB=3.␣␣ ␣␣
1 6 72

Source line: \ 1/(P+Q)**R LF XOff CR DE

FORTRAN line: ␣␣␣␣␣1/(P+Q)**R␣␣ ␣␣
1 6 72

Source line: 55 DE DE 555\␣X=Y+Z CR LF

FORTRAN line: 55555␣X=Y+Z CR LF␣␣ ␣␣
1 6 72

SEGMENTED PUNCHED TAPES

FORTRAN source programs that are input on paper tape may be punched on several pieces of tape, each of which is called a segment and which must have the four characters

:EOS

on a separate line at its end.

When the End of segment is read, the compiler waits for further input from the operaror's typewriter. The user can then remove the tape, put another segment in the reader and recommence reading by typing CR, LF .

A program module and a segment are entirely separate entities. A segment is physical and may contain several modules; conversely, a module may be split into several segments.

OPERATING PROCEDURES FOR PAPER TAPE INPUT

To load the compiler put the tape in the reader and type in the control command

LD

The compiler is read into memory and the system outputs the Monitor message

M:

The operator replies by typing

ST

which activates the compiler. Once started it outputs

F:

wich is request for the I/O options to be specified.
If the standard I/O options are to be used thc operator replies by typing

LF CR

(Standard I/O depends upon which devices have been assigned the standard file codes 1, 2 and 3 which specify the source input device, the listing output device and the object code output device respectively).

If other options are selected the operator must specify the file codes for the source input file, the listing output file and the object file plus, if required, the options Q and/or N, followed by LF CR.

Q specifies object code output in 4 x 4 format;
N indicates that the output listing is to be suppressed; this applies to all source modules being processed and has absolute priority over any OPTIONS statement in the source program. However, error messages will still be listed.

If object code output is not required, the code to suppress it is 0.
If the options are incorrectly typed, the compiler outputs a further

F:

and the options can be re-typed, ending with LF CR.

The compiler then begins reading and processing the source program, the first line of which must be an IDENT control statement. If the IDENT is missing the compiler will continue reading the source file, without processing, until the IDENT is found.

If :EOS is encountered in the input stream the compiler halts to allow a new tape segment to be loaded; reading is re-started by typing

LF CR

When an END statement is processed and is not followed by

:EOF - indicating further modules are to be compiled - the compiler punches

:EOS on the object code output file and, if no errors have been found in the END statement, outputs the message

F/MESSAGE

on the listing device and automatically continues reading a new source module (which must begin with an IDENT statement), without any operator action being necessary. If there is not another module, the next statement read must be :EOF

When :EOF is read in the input stream, the compiler checks that the last statement was an END statement, if it was not,

F/MESSAGE EM

is printed on the listing device and an END statement is automatically generated and compiled. The compiler then punches :EOF on the object code output file and types out

F:7F00 (indicating compilation successful)

A fresh compilation can then be started, or if no further processing is required, the operator types

:EOF

EXTENSIONS

Line Syntax

There is no limit imposed on the number of continuation lines.

The first character of any line may be the letter X which is used in the conditional compilation feature.

A comment line may be followed by a continuation line.

Subscripts

Any integer-valued expression is accepted as a subscript expression.

Arithmetic Expressions

Mixed mode arithmetic expressions are allowed.

DO Statement

The control variable, the initial, terminal and incrementation values may be redefined during execution of the range of the DO.
Only the incrementation value must be greater than zero.

Format Specifications

The FORMAT statement reference in a READ or WRITE statement may be specified by a label, an array name or a Hollerith constant which constitutes a valid format specification. When the format specification is in an array, an nH descriptor may form part of the specification.

Hollerith Descriptor

On output, the nH descriptor may be replaced by a string of characters enclosed within apostrophes. Data are then transferred exactly as written.

Control Statements

Two special statements - IDENT and OPTIONS - are available which control the compilation process.

Logical and Integer Expressions

Any integer-valued arithmetic expression may be used instead of a logical expression in any statement or expression, and conversely, any logical expression may be used instead of an integer expression.

RESTRICTIONS

In a statement function or assignment statement the = sign must be written in the initial line of the statement.

In any control statement the keyword (GOTO, CONTINUE etc.) must be written in the initial line of the statement; no part of it may be carried on to a continuation line.

The comma following the initial parameter in a DO statement must appear in the initial line of the statement.

Both STOP and PAUSE statements may be followed by a string of not more than four alphanumeric characters (as opposed to five in A.S.A).

The number of subscript expressions must not differ from the number of dimensions declared for an array. However, this does not prevent a two- or three-dimensional array from being made equivalent to a one-dimensional array (by application of the element successor rule).