Rdok 10414

Burroughs

B 6000/B 7000 Systems FORTRAN 77

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A.

October 1981

5011729

Burroughs

Hans Nems

B 6000/B 7000
Systems
FORTRAN 77

REFERENCE MANUAL

PRICED ITEM

Correspondence regarding this document should be addressed directly to Burroughs Corporation, Box CB7, Malvern, PA. 19355, Attn: Systems Documentation Dept., TIO East.

LIST OF EFFECTIVE PAGES

Title	Original
ii	Original
iii	Original
iv	Blank
v thru viii	Original
1-1 thru 1-5	Original
1-6	Blank
2-1 thru 2-12	Original
3-1 thru 3-5	Original
3-6	Blank
4-1 thru 4-10	Original
5-1 thru 5-3	Original
5-4	Blank
6-1 thru 6-3	Original
6-4	Blank
7-1 thru 7-12	Original
8-1 thru 8-23	Original
8-24	Blank
9-1 thru 9-18	Original
10-1	Original
10-2	Blank
11-1 thru 11-20	Original
12-1	Original
12-2	Blank
13-1 thru 13-11	Original
13-12	Blank
14-1 thru 14-6	Original
15-1 thru 15-12	Original
A-1 thru A-30	Original
B-1 and B-2	Original
1 thru 9	Original
10	Blank

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT)

TABLE OF CONTENTS (CONT)

LIST OF ILLUSTRATIONS

LIST OF TABLES

INTRODUCTION

This manual describes the Burroughs B 6000/B 7000 implementation of FORTRAN 77. FORTRAN 77 is the designation for the American National Standard Programming Language FORTRAN, ANSI X3.9-1978. Throughout this manual, any Burroughs extensions to this standard are shaded.

Specifically, this document describes both the programming language accepted by the B 6000/B 7000 FORTRAN 77 compiler and various options and control statements used with this compiler.

The manual is logically divided into two distinct parts. The first part consists of a discussion of the features of the B 6000/B 7000 system FORTRAN 77 language, while the second part discusses the FORTRAN 77 compiler. Sections 1 through 12 describe the language and it is to these sections that the user is referred for programming information. Sections 13 through 15 are devoted to a description of the compiler and contain information as to the manner in which FORTRAN 77 programs are submitted to the compiler and the extent to which the user may control compiler operation.

Because the basic features of the FORTRAN 77 language have been standardized and documented elsewhere, this document is concerned primarily with providing information concerning the B 6000/B 7000 systems implementation of this language. The first part of this document is devoted to a description of the constructs of FORTRAN 77 as implemented on the B 6000/B 7000 series system. The discussion proceeds from basic language elements to general FORTRAN 77 program structures. The pertinent sections contain the following information:

Section 1: Source input format and program structure.
Section 2: Constants, variables, arrays, and substrings; the primary units of which the language is constructed.
Sections 3 through 7, and 12: FORTRAN 77 expressions and statements; the computation-directed elements of the language.
Sections 8 and 9: I/O-related constructs.
Section 10: Main program.
Section 11: FORTRAN 77 subprograms and intrinsics.
Section 13: Storage allocation for program data items.
Sections 14 and 15: Compiler information and options.

In addition, appendices at the end of this manual provide reference aids for the programmer. Appendix A, which contains diagrams for the syntax of the language should be especially useful to the programmer as a quick programming reference.

SECTION 1

FORTRAN 77 TERMS AND CONCEPTS

NOTATIONS USED IN THIS MANUAL

In describing the form of FORTRAN 77 statements or constructs the following metalanguage conventions are used:

Upper case letters and words
These are mandatory and are to be written as shown.

Lower case letters and words
These indicate general entities for which specific user supplied entities must be substituted in the actual statements.

Brackets [ ]. These indicate an optional item.

Ellipsis,... This indicates that the item preceding the ellipsis may appear one or more times in succession.

Special characters Special characters from the FORTRAN 77 character set should be written as shown.

Blanks Used to improve readability, but unless otherwise noted have no significance.

Shaded areas denote Burroughs extensions.

Example:

Given the description of the form of a statement as:

```
CALL sub [([a[,a]...])]
```

the following forms are allowed:

```
CALL sub
CALL sub ()
CALL sub (a)
CALL sub (a,a)
CALL sub (a,a,a)
etc
```

When an actual statement is written, specific entities are substituted for sub and each a; for example,

```
CALL XYZ (A,1.0)
```

SYNTACTIC ITEMS

The basic syntactic items of the FORTRAN 77 language are constants, symbolic names, statement labels, keywords, operators, and special characters. Letters, digits, and special characters are used to form the syntactic items.

Constants	The forms of a constant are described in section 2.
Symbolic name	A symbolic name is a sequence of one to six letters or digits, the first of which must be a letter.
Statement label	A statement label consists of one to five digits, one of which must be nonzero. A statement label is used to identify a statement.
Keyword	A keyword is a specified sequence of letters. The keywords of the FORTRAN 77 language are described in sections 4 through 12. Whether a particular sequence of characters identifies a keyword or a symbolic name is implied by context.

The set of special characters is described in this section. A special character may be an operator or part of a constant or have some other special meaning. The interpretation is implied by context.

FORTRAN CHARACTER SET

The FORTRAN 77 compiler accepts source input coded in EBCDIC characters. The FORTRAN 77 character set consists of 26 letters, 10 digits, and 16 special characters.

A letter is one of the 26 characters:

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

A digit is one of the 10 characters:

0 1 2 3 4 5 6 7 8 9

A special character is one of 16 characters.

	Blank
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Decimal Point
$	Currency Symbol
'	Apostrophe
:	Colon
&	Apersand
%	Percent Sign
"	Quote

An alphanumeric character is a letter or a digit.

BASIC TERMS AND CONCEPTS

Certain basic concepts concerning the FORTRAN 77 language are presented here preliminary to the following discussion of B 6000/B 7000 implementation of this language.

Every FORTRAN 77 program consists of one or more program units including a required main program and possibly additional subprograms.

Each program unit is constructed of a series of statements. These statements specify the arithmetic operations which are to be executed, control the order in which program statements are to be performed, accomplish various program input and output functions, describe program data items, or provide other program information.

STATEMENTS

A statement is written in one or more lines, the first of which is called the initial line; succeeding lines, if any, are called continuation lines. There is also a line called a comment line, which is not part of any statement and is intended to provide documentation.

Unless the option FREE is set (section 15) FORTRAN 77 lines conform to the following formats:

1. A comment line is any line that contains a C or an asterisk in column 1, or contains only blank characters in columns 1 through 72. Comment lines may appear anywhere in the program unit.

2. An initial line is any line that is not a comment line and contains a blank or the digit 0 in column 6. Columns 1 through 5 may contain a statement label.

3. A continuation line contains any FORTRAN 77 character other than a blank or the digit 0 in column 6 and contains only blank characters in column 1 through 5. A statement must not have more than 19 continuation lines.

4. Columns 73 through 80 may contain identification or sequencing information.

5. A line containing a "$" in column 1 or a blank in column 1 and a "$" in column 2 is a compiler control card.

An END statement is written only in columns 7 through 72 of an initial line. No other statement in a program unit may have an initial line that appears to be an END statement. Note that a statement must not contain more than 1320 characters. Except as part of a logical IF statement, no statement may begin on a line that contains any part of the previous statement.

ORDER OF STATEMENTS

The order of statements in a main program or subprogram is determined by the following rules:

1. A PROGRAM statement may appear only as the first statement of a main program.

2. The first statement of a subprogram must be either a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

3. FORMAT statements may appear anywhere in a program unit.

4. File declaration statements must precede the first FORTRAN 77 statement.

5. All specification statements must precede all DATA statements, statement function statements, and executable statements.

6. Within the specification statements IMPLICIT statements must precede all other specification statements except PARAMETER statements. Any specification statement that specifies the type of a symbolic name of a constant must precede the PARAMETER statement that defines the constant.

7. All statement function statements must precede all executable statements.

8. Data statements may appear anywhere after specification statements.

9. Entry statements may appear anywhere except between a block IF statement and its corresponding END IF statement, or between a DO statement and the terminal statement of its DO-loop.

10. The last line of a program unit must be an END statement. END statements may not be continued.

Figure 1-1 is a diagram of the required order of statements and comment lines for a program unit. Vertical lines delineate varieties of statements that may be interspersed . For example, FORMAT statements may be interspersed with statement function statements. Horizontal lines delineate varieties of statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements. Note that an END statement is also an executable statement and must appear only as the last statement of a program unit.

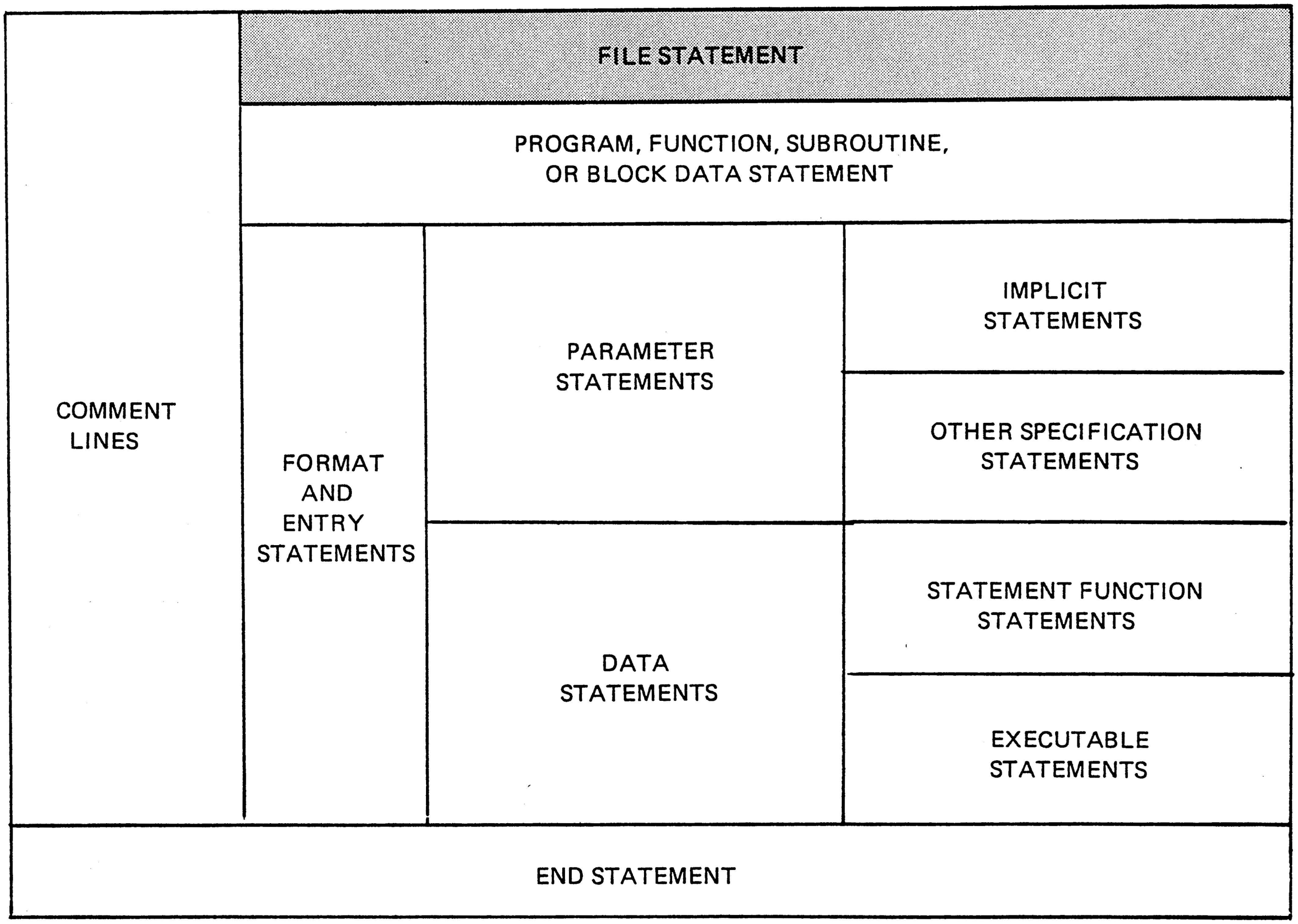

Figure 1-1. Order of Statements and Comment Lines

SECTION 2

CONSTANTS, VARIABLES, ARRAYS, AND SUBSTRINGS

CONSTANTS

Constants function as FORTRAN 77 data items and can be of the following types:

Integer
Real
Double Precision
Complex
Logical
Character
Hollerith
Hexadecimal

The value of a constant does not change.

INTEGER CONSTANTS

The format for an integer constant is:

sn

where s is an optional sign character (+ or -) and
n is a string of decimal digits representing an integer not greater than 549755813887.

The following are examples of valid integer constants:

0	9999999999	
+0	03770	
-0	1254	
17711	-5 7	(interpreted as -57)

The following are examples of invalid integer constants:

1.0	Decimal point not permitted; interpreted as a real constant.
3,000	No commas or other punctuation permitted.
1548009000000	Maximum value exceeded.

REAL CONSTANTS

The formats for a real constant are:

```
sm.n
smpnEx
```

where s is an optional sign character (+ or -),
m,n are strings of decimal digits with a combined total not exceeding 11 significant digits (leading zeros are not significant digits), one of which (but not both) may be omitted,
p is an optional decimal point, and
x is an integer constant.

The second form of a real constant is called the scientific or exponential form. The value represented is the number preceding the E multiplied by 10 raised to the power denoted by the integer constant following the E. Thus, the real constant 3E2 represents the value 300, which is 3 times 10 to the second power. If the decimal point is omitted, it is assumed to immediately precede the E. If the number of significant digits exceeds 11, approximately 11 significant digits will be retained.

If either form of real constant has no sign, it will be interpreted as representing a non-negative value.

The maximum absolute value of a real constant that is permitted is approximately:

4.3135914667 E68=((8**13)-1)*8**63

The smallest nonzero absolute value permitted is approximately:

8.7581154021 E-47=8**(-51).

Zero is also permitted.

The following are examples of valid real constants:

3.1415926535	2.5E+07 (same as 25000000)
0.	1.023342E+8
0.0	23456789012E-22
2E3	0000000000007.
2.E3	.5 3E-4 2 (interpreted as .53E-42)
2E+3	-253.
.075	-.075
6.02E23	2.9979E08

The following are examples of invalid real constants:

-1597	No decimal point or E portion; interpreted as an integer constant.
6.2E+68	Exceeds maximum size limit.
6.2E-48	Smaller than minimum size limit.
E22	Exponent part alone not permitted; interpreted as a variable name.
2.7E1.2	Exponent part must be an integer.
1E2E3	Only one E portion allowed per constant.
2,765,987.	No commas or other punctuation, except decimal point, permitted.

The internal machine representation of a real constant is similar to the internal representation of an integer constant. Thus, real and integer constants are often interchangeable in programs. However, care should be taken when intermingling such constants in expressions because the types employed in an expression affect the type of the expression.

DOUBLE PRECISION CONSTANTS

The formats for a double precision constant are:

supvDx

where s is an optional sign character,
u,v are strings of decimal digits, one of which (but not both) may be omitted,
p is an optional decimal point, and
x is an integer constant.

If the number of significant digits in a double precision constant exceeds 22, approximately 22 significant digits will be retained. The maximum absolute value of a double precision constant that is permitted is approximately:

1.94882838205028079124469 D+29603 = ((8**26)-1)*8**32754

The smallest nonzero absolute value permitted is approximately:

1.9385458571375858335564 D-29581 = 8**(-32755).

Zero is also permitted.

The following are examples of valid double precision constants:

```
1234567890.1234567890123 D 29594
1  2D-2 4259
6.63D-034
9.80665D+0
```

The following three are equivalent:

```
3.1415926535897932
3.1415926535897932D0
3.1415926535897932D-0
```

The following four are equivalent:

```
1D3
+1D+003
+1.D+3
1.D0003
```

The following are examples of invalid double precision constants:

3.14159265	No D portion; interpreted as a real constant.
2.7 D29604	Exceeds maximum size limit.
2.7 D-29585	Smaller than minimum size limit.
444	Integer constant.
1.234,567,890,123.	Commas not permitted.
1.3E456	No D in exponent part.

COMPLEX CONSTANTS

The format for a complex constant is:

(m,n)

where m and n are the real and imaginary part, respectively, each of which may be an integer or real constant.

The complex constant (m,n) represents the quantity m+ni, where i equals the square root of -1. The following are examples of valid complex constants with the equivalent mathematical expressions beside them:

Complex Constant	Equivalent Mathematical Expressions
(123,456)	123 + 456i
(3.14159265,1597)	3.14159265 + 1597i
(0,.05)	0 + 0.05i or 0.05i
(100.,100.00)	100 + 100i
(1E2,0)	100 + 0i or 100
(0,-1)	-i

The following are examples of invalid complex constants:

(3,)	Missing imaginary part.
(12345.D10,0)	Neither part may be a double precision constant.
(12.0,72.E68)	Imaginary part exceeds size limit for real constants.
(3,4,5)	Excessive number of parts - more than two.

LOGICAL CONSTANTS

Two logical constants are provided to represent the logical values "true" and "false".

These two logical constants have the form:

.TRUE.
.FALSE.

The use of these logical constants is restricted to certain types of expressions (see discussion of logical expressions.) The internal machine representations of these two constants are such that the data words corresponding to the constant TRUE and the integer constant 1 are identical and the data corresponding to the constant FALSE and the integer constant 0 are identical.

CHARACTER CONSTANTS

The format for a character string is:

'S'
"S"

where s is a nonempty string of no more than 255 characters.

The apostrophe or the quote may be used as the string delimiter. If the string delimiter is the apostrophe then the apostrophe is interpreted as part of the string only if two adjacent apostrophes appear in the string in its place. These two apostrophes are treated as representing one apostrophe character. If the quote is the string delimiter then two consecutive quotes in a string are interpreted as representing a single quote character.

The length of the character constant is the number of characters between delimiting apostrophes or quotes except that each pair of consecutive delimiters counts as a single character. Blanks embedded between delimiters are significant.

Each character in the string has a character position that is numbered consecutively 1, 2, 3 and so on. The number indicates the sequential position of a character in the string beginning at the left and proceeding to the right.

The following are examples of valid character constants:

```
"ABCDEFG" stored as ABCDEFG
'EFGH' stored as EFGH
'A"BC' equivalent "A""BC" stored as A"BC
'A''BC' equivalent "A'BC" stored as A'BC
```

HOLLERITH CONSTANTS

The format for a Hollerith string is:

```
wHs
```

where w is a positive integer constant denoting the width of the string in characters and
s is a string containing no more than 255 characters.

The use of hollerith constants is restricted to Format statements.

The following are examples of valid Hollerith constants:

```
2HbQ
4H"C'.
7Hbb"ABC"
```

The following are examples of invalid Hollerith constants:

3HbbbAB	Character string may not be longer than the integer preceding the H indicates.
-6HABBCDEF	String width specification may not be a negative.
OHA	String width specification may not be zero.

HEXADECIMAL CONSTANTS

Another alternate representation of program values is the hexadecimal constant literal which corresponds to notation in a number system with a radix of 16.

The format for a hexadecimal constant is:

```
Zn
```

where n is a string of hexadecimal digits.

The hexadecimal digits 0,1,...., 9 are represented by the characters 0,1,...,9 respectively, and the hexadecimal digits equal in value to 10,11,...,15 are represented by the letters A,B,..,F respectively. Hexadecimal strings are stored right justified with leading binary zeros appended to fill the word. If the string contains more digits than will fit in a word then the left-most digits of the string will be retained.

The following are examples of valid hexadecimal constants with their decimal equivalents:

Hexadecimal Constant	Decimal Equivalent
Z0	0
Z10	16
Z100	256
ZF F	255

The following are examples of invalid hexadecimal constants:

FFF60	Missing Z.
ZABCDGF	Contains the character G which is not hexadecimal digit.
Z100200300400A0	Contains more than 12 digits.
Z333.33 003	Contains a decimal point which is illegal for a hexadecimal constant.

A hexadecimal constant may only be used for data initialization in DATA and type statements.

VARIABLES

A variable is an entity which has a symbolic name and type. The value of the datum represented by a variable may be altered during program execution.

A variable identifies one or more storage locations for purposes of data storage and retrieval. The contents of these storage locations are accessed by referencing the associated variable name.

Note that the term variable refers to a simple variable and does not include array elements.

A variable name is an identifier which consists of one to six alphanumeric characters (such as letter or digit characters), with the leading character being a letter. If the variable name is more than six characters long, then only the first six characters of this identifier will be retained internally. Embedded blanks are ignored.

Variables may be classified into six types:

Integer
Real
Double precision
Complex
Logical
Character

The value represented by a variable of each of these types may be expressed by a constant of the same type. The values represented by each variable type must, therefore, obey the magnitude and significant- digit restrictions governing the corresponding type of constant. A variable is assigned a type according to the initial character of its name unless declared otherwise in a specific type statement or an IMPLICIT statement. If the initial character is the letter I,J,K,L,M, or N, then the variable will be of type integer. If the initial character is any other letter, then the variable will be of type real. No such defaults exist for types double precision, complex, logical, or character. Variables of these types must be declared as such by appropriate type statements or IMPLICIT statements.

The following are examples of valid variable names with the type assigned to them when they are not affected by type or IMPLICIT statements:

AMK599	Type real.
X64000	Type real.
OF TEN	Type real; interpreted as OFTEN.
LOOP3	Type integer.
REALNUMBER	Type real;truncated to six characters and stored internally as REALNU.

The following are examples of invalid variable names:

3LOOP	Variable name may not begin with a digit character.
BE-GIN	Characters other than letters, digits, or blanks are not allowed in a variable name.

ARRAYS

An array is an ordered set of data organized into dimensions. An array is referenced by an array name which is of the same form as a variable name. A variable name is designated as an array name by its appearance in an array declarator. All of the elements of an array are of the same type, and this type is indicated by the array name in the same manner as for variable names.

An array element name identifies one element of the array. An array element name is an array name qualified by a subscript.

Following is the format of an array element name:

```
a(s[,s]...)
```

where a is an array name, and
s is a subscript expression.

The number of subscript expressions must be equal to the number of dimensions in the array declarator for the array. A subscript expression is an integer expression. The value determines which element in the array is identified by the array element name.

An array name may appear without a subscript in the following cases:

1. The dummy argument list of a subprogram.

2. The actual argument list of a subprogram reference.

3. An I/O list if the array is not an assumed-size dummy array.

4. A COMMON, EQUIVALENCE, DATA, or type statement.

5. A READ, WRITE, PRINT, or PUNCH statement as the format identifier if the array is not an assumed-size dummy array.

6. A READ or WRITE statement as the file designator.

7. A SAVE statement.

8. An I/O statement as a unit identifier for an internal file if the array is not an assumed-size dummy array.

ARRAY DECLARATORS

An array declarator declares the size and number of dimensions of an array.

The format for an array declarator is:

```
a(d[,d]...)
```

where a is an array name, and
d is a dimension declarator.

The number of dimensions in the array is the number of dimension declarators in the array declarator. The minimum number of dimensions is one and the maximum is 7.

The format for a dimension declarator is:

[d1:]d2

where d1 is the lower dimension bound and
d2 is the upper dimension bound.

The lower and upper dimension bounds are integer arithmetic expressions. The value of a dimension bound may be positive, negative or zero; however, the value of the upper dimension bound must be greater than or equal to the value of the lower dimension bound. If only the upper dimension bound is specified, the value of the lower dimension bound is 1. An array declarator must precede the first use of that array in an executable, DATA or EQUIVALENCE statement.

Each array declarator is either a constant array declarator, an adjustable array declarator or an assumed-size array declarator. A constant array declarator is an array declarator in which each of the dimension bounds is a constant. An adjustable array declarator is an array declarator that contains one or more variables. An assumed-size array declarator is constant array declarator or an adjustable array declarator, except that the upper bound of the last dimension is an asterisk.

In addition, each array declarator is either an actual array declarator or a dummy array declarator. An actual array declarator is an array declarator in which the array name is not a dummy argument. An actual array declarator must be a constant array declarator and is permitted in DIMENSION statements, COMMON statements and type statements. A dummy array declarator is an array declarator in which the array name is a dummy argument. A dummy array declarator may be either a constant array declarator, an adjustable array declarator, or an assumed-size array declarator. A dummy array declarator is permitted in DIMENSION statements and type statements, and may appear only in function or subroutine subprograms.

The size of a dimension is the value d2-d1+1, where d1 and d2 are the dimension bounds for that dimension. The size of an array is equal to the number of elements in the array and is equal to the product of the sizes of the dimensions specified by the array declarator for that array. In the case of an assumed-size dummy array the size is determined as follows:

1. If the actual argument corresponding to the dummy array is a noncharacter array name, the size of the dummy array is the size of the actual argument array.

2. If the actual argument corresponding to the dummy array name is a noncharacter array element name with a subscript value of r in an array of size x, the size of the dummy array is x + 1 -r.

3. If the actual argument is a character array name, character array element name, or character array element substring name and begins at character storage unit t of an array with c character storage units, then the size of the dummy array is INT((c+1-t)/ln), where ln is the length of an element of the dummy array.

If an assumed-size dummy array has n dimensions, the product of the sizes of the first n-1 dimensions must be less than or equal to the size of the array, as determined by one of the immediately preceding rules.

The following are examples of valid array elements:

AMK 599(6)	Interpreted as AMK599(6).
ZAP IT(3)	Interpreted as ZAPIT(3) when used where an array element may appear.
J5(J5(3))	The subscript is itself an array element.
ARRAY2 (1,1,1,1)	
A(R*T)	
INTT(1+9*ISINO*IP110)	

The following are examples of invalid array elements:

A(A)	A subscript must be a valid arithmetic expression; an array name does not constitute such an expression.
3ARRAY(6)	An array name must be a variable name.

SUBSTRINGS

A substring is a contiguous portion of a character datum. It is of type character. A character substring is identified by a substring name and may be assigned values and referenced.

The forms of a substring name are:

```
v ( [c] : [d] )
a (s [,s]...) ( [c] : [d] )
```

where a (s [,s]...) is a character array element name,
v is a character variable name, and
c and d are each an integer expression and are called substring expressions.

The value c specifies the leftmost character position of the substring, and the value d specifies the rightmost character position. For example A(2:4) specifies characters in positions 2 through 4 of the character variable A, and B(4,3) (1:6) specifies characters in positions 1 through 6 of the character array element B(4,3). The values of c and d must be such that 1 <= c <= d <= len, where len is the length of the character variable or array element. If c is omitted, a value of one is implied for c. If d is omitted, a value of len is implied for d. Both c and d may be omitted; for example, the form v(:) is equivalent to v, and the form a(s[,s]...)(:) is equivalent to a(s [,s]...). The length of a character substring is d - c + 1.

A substring expression may be any integer expression. A substring expression may contain array element references and function references. Evaluation of a function must not alter the value of any other expression within the same substring name.

The following are examples of substrings:

```
AB(5:6)
CX(5)(2:1)
G(5:)
ZB(:)
GY(:JK*3)
```

SECTION 3

EXPRESSIONS

INTRODUCTION

An expression consists of operands, operators, and parentheses. An operand is a valid constant, variable, function or intrinsic reference, or expression enclosed in parentheses. The expression represents the value obtained when the indicated operations are performed on the indicated values. There are four types of expressions: arithmetic, character, relational, and logical.

OPERATORS

There are four types of operators which may be used to form FORTRAN77 expressions. They are arithmetic operators, relational operators, logical operators, and character operators.

The five arithmetic operators are as follows:

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

The precedence among the various operators which determines the order in which the operands are to be combined unless the order is changed by parentheses is as follows from highest to lowest:

**	Highest
* and /	Intermediate
+ and -	Lowest

There is only one character operator concatenation, denoted by parallel slashes.

//	Concatenation

The six relational operators are of equal precedence and are as follows:

.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 five logical operators are as follows:

.NOT.	Logical Negation
.AND.	Logical Conjunction
.OR.	Logical Inclusive Disjunction
.EQV.	Logical Equivalence
.NEQV.	Logical Nonequivalence

The precedence among the various operators which determines the order in which the operands are to be combined unless the order is changed by parentheses from highest or lowest is as follows:

.NOT.	Highest
.AND.	
.OR.	
.EQV. or .NEQV	Lowest

The precedence among the various types of operators is as follows:

Arithmetic	Highest
Character	
Relational	
Logical	Lowest

Operations of equal precedence are performed from left to right, except exponentiation which is carried out from right to left. Parentheses may be used to override operator precedence in the usual manner. The unary + operator is ignored. Note that adjacent operators are not permitted.

ARITHMETIC EXPRESSION

An arithmetic expression is used to express a numeric computation. Evaluation of an arithmetic expression produces a numeric value. An arithmetic expression may contain only arithmetic operators, symbolic names of arithmetic constants, arithmetic constants, variables, arithmetic functions and intrinsic function references, and grouping parentheses.

The types of operands in an arithmetic expression determine the type of the value obtained from the evaluation of the expression. The resultant types of arithmetic operations depend upon the types of the operands and the operator involved. "DOUBLE" indicates type double precision.

For the operators +, -, *, and /, the result of the operation will be of the following type:

Type of First Operand	Type of Second Operand: Integer	Real	Double	Complex
Integer	INTEGER	REAL	DOUBLE	COMPLEX
Real	REAL	REAL	DOUBLE	COMPLEX
Double	DOUBLE	DOUBLE	DOUBLE	COMPLEX
Complex	COMPLEX	COMPLEX	COMPLEX	COMPLEX

For exponentiation (**) the type and the interpretation of the expression X1**X2 is a follows:

X2 / X1	I2	R2	D2	C2
I1	I =I1**I2	R =REAL(I1)**R2	D =DBLE(I1)**D2	C =CMPLX(REAL(I1),0.)**C2
R1	R =R1**I2	R =R1**R2	D =DBLE(R1)**D2	C =CMPLX(R1,0.)**C2
D1	D =D1**I2	D =D1**DBLE(R2)	D =D1**D2	Prohibited
C1	C =C1**I2	C =C1**CMPLX(R2,0.)	Prohibited	C =C1**C2

Four entries shown previously specify an interpretation to be a complex value raised to a complex power. In these cases, the value of the expression is the "principal value" determined by X1**X2 = EXP (X2*LOG(X1)), where EXP and LOG are functions described in the function section of this manual.

If the value of I2 is negative, the interpretation of 1/(I1**ABS(I2)) is subject to the rules for integer division. For example, 2**(-3) has the value of 1/(2**3), which is zero.

The following are examples of valid arithmetic expressions:

```
6
J+6
XALPHA(8)/A + (ZZ + XALPHA(9))/ZETA
LBD/(A(1)+A(2))
20**FCN1(XND(1,2),12)
-B*A
-(-P)
```

The following are examples of invalid arithmetic expressions:

B**-A	Illegal adjacent operators
Q,R	Illegal character
2+A.AND.B	Illegal logical operator
(A+B) (C+D)	Illegal implied multiplication

CHARACTER EXPRESSION

A character expression is used to express a character string. Evaluation of a character expression produces a result of type character.

A character expression may contain character constants, symbolic names of character constants, character variables, character array elements, character substrings, and character functions combined with character operators and parentheses. Concatenation, denoted by parallel slashes, is the only character operator.

The interpretation of the expression formed with the character operator is:

Use of Operator	Interpretation
x1 // x2	Concatenate x1 with x2

where: x1 denotes the operand to the left of the operator
x2 denotes the operand to the right of the operator

The result of a concatenation operation is a character string whose value is the value of x1 concatenated on the right with the value of x2 and whose length is the sum of the lengths of x1 and x2. For example, the value of 'AB' // 'CDE' is the string ABCDE. Except in a character assignment statement, a character expression must not involve concatenation of a character operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant.

The following are examples of character expressions and their values. C is a character variable of length 3 which has been assigned the value "ABC", and D is a character variable of length 4 which has been assigned the value "DEFG".

EXPRESSION	VALUE
"AB"	"AB"
"FG" // "HI"	"FGHI"
C // "DE"	"ABCDE"
"AB" // D(1:2)	"ABDE"
"A" // C(2:) // D	"ABCDEFG"

RELATIONAL EXPRESSION

A relational expression is used to compare the values of two arithmetic expressions or two character expressions. The value of an arithmetic expression may not be compared to a character expression. A relational expression consists of two expressions both either arithmetic or character separated by one of the six relational operators. Evaluation of the expression produces the logical value of true or false. It is interpreted as having the logical value true if the values of the operands satisfy the relation specified by the operator, and false if they do not satisfy the relation. Only .EQ. or .NE. are permitted with complex operands. The comparison of a double precision value and a complex value is not permitted.

When the operands of the relational expression are of type character they are compared as to their relative value in the EBCDIC collating sequence. If the character operands are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

Examples of relational expressions follow. In these examples A,B and C are real variables. D is a double precision variable. C1 and C2 are complex variables, and CA, CB, and CD are character type variables.

```
B+A.GT.D
C1.EQ.C2
CA//CB.GE.CD
CA(2:3).LT. CB
A*B/9.2.LE. C
C1.NE. A
```

LOGICAL EXPRESSION

A logical expression represents a rule for computing a value corresponding to .TRUE. or .FALSE. and consists of any valid combination of logical operands, logical operators, and parentheses. A logical operand may be a logical constant, symbolic name of a logical constant, logical variable, logical array element, logical function reference, a relational expression, or a logical expression in parentheses.

Logical negation is expressed by the operator .NOT.. The operator .NOT. complements the value of a logical operand.

The .AND. operator is used to express the logical product of two operands. The operation A.AND.B, yields the value .TRUE. if and only if both A and B are .TRUE.. Otherwise the value of A.AND.B is .FALSE..

The .OR. operator is used to express the logical sum of any two logical operands. The operation A.OR.B, yields the value .FALSE. only if both A and B are .FALSE.. Otherwise, the value of A.OR.B is .TRUE..

The .EQV. operator is used to express the logical equivalence of two logical operands. The operation yields the value .TRUE. if both A and B are .TRUE. or both A and B are .FALSE.. Otherwise the value of A.EQV.B is .FALSE..

The .NEQV. operator is the negation of the logical equivalence. The operation A.NEQV.B yields the value .TRUE. if and only if one of the operands A or B is .TRUE.. Otherwise, the value of A.NEQV.B is .FALSE..

The following are examples of logical expression: A, B, and C are real variables; L1, L2, and array L3 are logical variables; TRU is the symbolic name of a logical constant; and EVAL is a logical function.

```
A+B.GT.C.OR. L2
.NOT.(L1.OR.L3).AND.TRU
EVAL(A).NEQV. L1
L3(3).OR.L2 .EQV. L1.AND. L3(2)
```

SECTION 4

SPECIFICATION STATEMENTS

INTRODUCTION

The nonexecutable specification statements are employed to supply compile-time information about the types, initial values, and storage allocation of variables and arrays, and to allow subprograms to be used as actual arguments.

The specification statements include the following:

1. Dimension
2. Equivalence
3. Common
4. Type statements - INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL and CHARACTER
5. Implicit
6. Parameter
7. External
8. Save
9. Intrinsic

DIMENSION STATEMENT

The nonexecutable DIMENSION statement specifies the name and dimensions of an array.

The format for the DIMENSION statement is:

```
DIMENSION a(d) [,a(d)]...
```

where a(d) is an array declarator.

The format of an array declarator is specified in section 2.

Each array referenced in a program unit must have its array bounds specified exactly once in an array declarator in that program unit. Array declarators may appear in COMMON and type statements in addition to the DIMENSION statement.

Examples:

```
DIMENSION D(2:3), E(2:5,-1:0)
DIMENSION A(10)
DIMENSION B(N,2),C(6)
DIMENSION Q(J)
```

EQUIVALENCE STATEMENT

An EQUIVALENCE statement is used to specify the sharing of memory locations by two or more entities in a program unit. If the equivalenced entities are of different data types, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence.

The format for the EQUIVALENCE statement is:

```
EQUIVALENCE (nlist) [,(nlist)]....
```

where nlist is a list of 2 or more variable names, array element names, array names, and character substring names.

Names of dummy arguments of an external procedure in a subprogram must not appear in the list. If a variable name is also a function name, that name must not appear in the list.

Each subscript expression or substring expression in a list must be an integer constant expression.

An entity of type character may be equivalenced only with other entities of type character. The lengths of the equivalenced entities are not required to be the same. If an array element name appears in an EQUIVALENCE statement, the number of subscript expressions must be the same as the number of dimensions specified in the array declarator for the array name.

The use of an array name unqualified by a subscript in an EQUIVALENCE statement has the same effect as using an array element name that identifies the first element of the array.

An EQUIVALENCE statement must not specify that the same storage unit is to occur more than once in a storage sequence, as shown in the following example:

```
DIMENSION A(2)
EQUIVALENCE (A(1),B), (A(2),B)
```

This is prohibited, because it would specify the same storage location for A(1) and A(2). An EQUIVALENCE statement must not specify that consecutive storage units are to be nonconsecutive. For example, the following is prohibited:

```
REAL A(2)
DOUBLE PRECISION D(2)
EQUIVALENCE (A(1), D(1)), (A(2),D(2))
```

Following are valid examples of the equivalence statement:

```
DIMENSION X(5), Z(10)
EQUIVALENCE (X,Z(6),Y)
```

Y,X(1) and Z(6) all share the same memory location. Z(7) through Z(10) share locations with X(2) through X(5).

```
CHARACTER A*4, B*4, C(2)*3
EQUIVALENCE (A,C(1)), (B,C(2))
```

The association of A, B, and C can be graphically illustrated as:

```
↑01↑02↑03↑04↑05↑06↑07↑
↑-----A-----↑
↑--C(1)--↑-----B-----↑
         ↑--C(2)--↑
```

COMMON STATEMENT

The nonexecutable COMMON statement associates variables and arrays with blocks of storage which may be shared among several program units.

The format for the COMMON statement is:

```
COMMON [/[n]/] v [[,]/[n]/v]...
```

where n is an optional common block name, and
v is a list of variables, array names or array declarators. No dummy arguments may appear in the list v.

A symbolic name is associated with each block of common storage; this name is called a common block name. Any program unit may access the block of storage associated with this name via a common statement employing this name. Common storage associated with a common block name is called named common. A common block name is constructed in the same manner as a variable name. This name may not be a dummy argument. A common block name is considered as such only in the context of a COMMON statement. A common block need not be named; common storage associated with no name is called blank common.

The following statements are equivalent and define a blank common block:

```
COMMON//A,B(10)
COMMON A,B(10)
```

Variables and arrays are assigned contiguous locations in common storage in the order in which they appear in a COMMON statement. Entire arrays but not individual array elements may be assigned storage locations in common storage.

If a character variable or character array is in a common block, all of the entities in that common block must be of type character.

If the same common name appears more than once in a program unit, the common elements associated with a second or subsequent appearance are considered as extensions to the list of the previous appearances.

The EQUIVALENCE statement may be used to associate elements with a common block. This association may extend the common block beyond its former terminal point, thus increasing the size of the common block. Such an association may not extend a common block backwards to locations preceding its initial point. Two elements of common storage may be made equivalent to one another, either directly or indirectly, by an EQUIVALENCE statement.

Variables and array names may not be duplicated in COMMON statements. One variable or array may not be assigned to more than one block of common storage within a program unit.

Following are examples of the COMMON statement:

```
COMMON/BLOCK1/A,B(10),C//G,HOLD,/BLOCK2/Q(3)
COMMON C
COMMON T1/CMN/T2,T3
```

TYPE STATEMENT

A type statement is used to override or confirm implicit typing and may specify dimension information.

The appearance of the symbolic name of a constant, variable, array, or function in a type statement specifies the data type for that name for all appearances in the program unit. Within a program unit, a name must not have its type explicitly specified more than once.

The format for Integer, Real, Double Precision, Complex, or Logical type statements is:

```
t m[/v/][,m[/v/]]...
```

where t is one of INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL,
m is a variable, array name, array declarator, or function name, and
v is an initial value list as specified for the data statement.

The format for a character type statement is:

```
CHARACTER [*len [,]] nam [/clist/][,nam [/clist/]]...
```

where clist is an initial value list of the form specified for the data statement, and
nam is of the form:

```
v[*len]
a[(d)][*len]
```

where v is a variable name, symbolic name of a constant, function name, or dummy procedure name,
a is an array name,
(d) is an array declarator,
len is the length (number of characters) of a character entity and called the length specification. len is one of the following:

1. An unsigned, nonzero, integer constant.
2. An integer constant expression enclosed in parentheses and with a positive value.
3. An asterisk in parentheses (*).

A comma is not required after a slash.

A length immediately following the word CHARACTER is the length specification for each entity in the statement not having its own length specification. A length specification immediately following an entity is the length specification for only that entity. Note that for an array the length specified is for each array element. If a length is not specified for an entity, its length is 1. An entity declared in a CHARACTER statement must have a length specification that is an integer constant expression, unless that entity is an external function, a dummy argument of an external procedure, or a character constant that has a symbolic name.

If a dummy argument has a length of (*) declared, the dummy argument assumes the length of the associated actual argument for each reference of the subroutine or function. If the associated actual argument is an array name, the length assumed by the dummy argument is the length of an array element in the associated actual argument array.

If an external function has a length of (*) declared in a function subprogram, the function name must appear as the name of a function in a FUNCTION or ENTRY statement in the same subprogram. When a reference to such a function is executed, the function assumes the length specified in the referencing program unit.

The length specified for a character function in the program unit that references the function must be an integer constant expression and must agree with the length specified in the subprogram that specifies the function. Note that there always is agreement of length if a length of (*) is specified in the subprogram that specifies the function.

If a character constant that has a symbolic name has a length of (*) declared, the constant assumes the length of its corresponding constant expression in a PARAMETER statement.

The length specified for a character statement function or statement function dummy argument of type character must be an integer constant expression.

INITIAL VALUE LISTS

The type statement may be used instead of the DATA statement for variable initialization. Type statements may optionally contain initial value lists constructed in the same fashion as the initial value list in a DATA statement. (See Data Statement for the syntax.)

A comma after a slash is not required.

Examples of type statements follow:

```
REAL INT,N3(5),IL1
INTEGER A1,R2/2,3/ A(3)/3*300/Z
CHARACTER*5,D,B,C*3
CHARACTER*2 G,H/"HA","HB"/,L,M
```

IMPLICIT STATEMENT

The IMPLICIT statement allows the default types assigned to variable names according to their initial characters to be altered.

The format for the IMPLICIT statement is:

```
IMPLICIT t(a[,a]...) [,t(a[,a]...)]...
```

where t is one of INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL or CHARACTER [*len],

a is either a single letter or a range of single letters in alphabetical order. A range is denoted by the first and last letter of the range separated by a minus. Writing a range of letters a1-a2 has the same effect as writing a list of the single letters a1 through a2 inclusive and

len is the length of the character entities and is one of the following:

1. An unsigned, nonzero, integer constant.
2. An integer constant expression enclosed in parentheses and with a positive value. If the length is not specified, the length is 1.

An IMPLICIT statement specifies a type for all variables, arrays, symbolic names of constants, and functions (except intrinsic functions) that begin with any letter that appears in the specification, either as a single letter or included in a range of letters. IMPLICIT statements do not change the type of any intrinsic functions. An IMPLICIT statement applies only to the program unit that contains it. Type specification by an IMPLICIT statement may be overridden or confirmed by the use of a type statement. An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the name of that function subprogram. Note that the length is also overridden when a particular name appears in a CHARACTER or CHARACTER FUNCTION statement.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER statements. A program unit may contain more than one IMPLICIT statement.

The same letter must not appear as a single letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a program unit.

Examples of the IMPLICIT Statement follow:

```
IMPLICIT REAL (I,L-N), LOGICAL (C,F)
IMPLICIT CHARACTER*3 (Z,X)
IMPLICIT DOUBLE PRECISION (B,D),CHARACTER(L)
```

PARAMETER STATEMENT

The PARAMETER Statement is used to give a constant a symbolic name.

The format of the PARAMETER statement is:

```
PARAMETER (p=e [,p=e]...)
```

where p is a symbolic name, and
e is a constant expression.

If the symbolic name is of type integer, real, double precision, or complex, the corresponding expression must be an arithmetic constant expression. If the symbolic name is of type character or logical, the corresponding expression must be a character constant expression or a logical constant expression, respectively.

The constant becomes defined with the value of the expression that appears on the right of the equals, in accordance with the rules for assignment statements.

Any symbolic name of a constant that appears in an expression must have been defined previously in the same or a different PARAMETER statement in the same program unit. A symbolic name of a constant must not become defined more than once in a program unit.

If a symbolic name of a constant is not of default implied type, its type must be specified by a type statement or IMPLICIT statement prior to its first appearance in a PARAMETER statement. If the length specified for the symbolic name of a constant of type character is not the default length of one, its length must be specified in a type statement or IMPLICIT statement prior to the first appearance of the symbolic name of the constant. Its length must not be changed by subsequent statements including IMPLICIT statements.

Once a constant is defined, it may appear in that program unit in any subsequent statement as a primary in an expression or in a DATA statement. A symbolic name of a constant must not be part of a format specification. A symbolic name of a constant must not be used to form part of another constant, for example, any part of a complex constant. A symbolic name in a PARAMETER statement may identify only the corresponding constant in that program unit.

Following are examples:

```
PARAMETER (PVALUE=5.6789)
PARAMETER (RFACTOR=3.563, R2FACTOR=2*RFACTOR, T=.TRUE.)
```

EXTERNAL STATEMENT

An EXTERNAL statement is used to identify a symbolic name as representing an external procedure or dummy procedure, and to permit such a name to be used as an actual argument.

The format for the EXTERNAL statement is:

```
EXTERNAL proc [,proc]...
```

where proc is the name of an external procedure, block data subprogram, or dummy procedure.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure name, dummy procedure name, or block data subprogram name. If an external procedure or a dummy procedure name is used as an actual argument in a program unit, it must appear in an EXTERNAL statement in that program unit. Note that a statement function name must not appear in an EXTERNAL statement.

If an intrinsic function name appears in an EXTERNAL statement in a program unit, that name becomes the name of some external procedure and an intrinsic function of the same name is not available for reference in the program unit.

The following are examples of valid EXTERNAL statements:

```
EXTERNAL EQUIV
EXTERNAL SUBA, FCN1
```

INTRINSIC STATEMENT

An INTRINSIC statement is used to identify a symbolic name as representing an intrinsic function. It also permits a name that represents a specific intrinsic function to be used as an actual argument.

The format for the INTRINSIC statement is:

```
INTRINSIC fun [,fun]...
```

where fun is an intrinsic function name.

Appearance of a name in an INTRINSIC statement declares that name to be an intrinsic function name. If a specific name of an intrinsic function is used as an actual argument in a program unit, it must appear in an INTRINSIC statement. The names of intrinsic functions for type conversion (INT, IFIX, IDINT, FLOAT, SNGL, REAL, DLE, CMPLX, ICHAR, CHAR), lexical relationship (LGE, LGT, LLE, LLT), for choosing the largest or smallest value (MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, MIN1), TIME, and DATE must not be used as actual arguments.

The appearance of a generic function name in an INTRINSIC statement does not cause that name to lose its generic property.

Only one appearance of a symbolic name in all of the INTRINSIC statements of a program unit is permitted. Note that a name must not appear in both an EXTERNAL and INTRINSIC statement in a program unit.

Following are examples:

```
INTRINSIC TANH
INTRINSIC LOG10, EXP
```

SAVE STATEMENT

A SAVE statement is used to retain the definition status of an entity after the execution of a RETURN or END statement in a subprogram. Within a function or subroutine subprogram, an entity specified by a SAVE statement does not become undefined as a result of the execution of a RETURN or END statement in the subprogram. However such an entity in a common block may become undefined or redefined in another program unit.

The form of a SAVE statement is:

```
SAVE [a [,a]...]
```

where a is a named common block name preceded and followed by a slash, a variable name, or an array name. Redundant appearances of an item are not permitted.

Dummy argument names, procedure names, and names of entities in a common block must not appear in a SAVE statement.

A SAVE statement without a list is treated as though it contained the names of all allowable items in that program unit.

The appearance of a common block name preceded and followed by a slash in a SAVE statement has the effect of specifying all of the entities in that common block. If a particular common block name is specified by a SAVE statement in a subprogram of an executable program, it must be specified by a SAVE statement in every subprogram in which that common block appears.

A SAVE statement is optional in a main program and has no effect. If a named common block is specified in a SAVE statement in a subprogram, the current values of the entities in the common block at the time a RETURN or END statement is executed are made available to the next program unit that specifies that common block name in the execution sequence of an executable program.

If a named common block is specified in the main program unit, the current values of the common block storage sequence are made available to each subprogram that specifies that named common block; a SAVE statement in the subprogram has no effect.

If a local entity that is specified by a SAVE statement and is not in a common block is in a defined state at the time a RETURN or END statement is executed in a subprogram, that entity is defined with the same value at the next reference of that subprogram.

The execution of a RETURN statement or an END statement within a subprogram causes all entities within the subprogram to become undefined except for the following:

1. Entities specified by SAVE statements.
2. Entities in blank common.
3. Initially defined entities that have neither been redefined nor become undefined.
4. Entities in a named common block that appears in the subprogram and appears in at least one other program unit that is referencing, either directly or indirectly, that subprogram.

Following are examples:

```
SAVE
SAVE /CMBLKA/,AX,ZZ,/CMBLKB/
```

SECTION 5

DATA STATEMENT

INTRODUCTION

The DATA statement is used to provide initial values for variables, arrays, array elements, and substrings. A DATA statement is nonexecutable and may appear in a program unit anywhere after the specification statements, if any. All initially defined entities are defined when an executable program begins execution.

The format of a DATA statement is:

```
DATA nlist / clist/ [[,] nlist / clist/]...
```

where nlist is a list of variables names, array names, array element names, substring names, and implied-DO lists.

clist is a list of the form:

a [,a]...

where a is one of the forms:

c

r*c

where c is a constant or the symbolic name of a constant

where r is a nonzero, unsigned, integer constant or the symbolic name of such a constant. The r*c form is equivalent to r successive appearances of the constant c.

Names of dummy arguments and functions must not appear in the list of elements to be defined. The names of entities in a common block may appear in this list only within a main program or a block data subprogram.

There is a one-to-one correspondence between the items specified by the list of elements, nlist, and the constants specified by the list of constants, clist, such that the first item of nlist corresponds to the first constant of clist, etc. By this correspondence the initial value is established and the entity is initially defined.

Correspondence between the entities in the element list and the constants in the constant list is restricted to the following:

1. If the nlist entity is of type integer, real, or double precision, the corresponding clist constant must be of type integer, real, double precision, or hexadecimal.

2. If the nlist entity is of type complex, the corresponding clist constant must be of type integer, real, double precision, complex, or hexadecimal.

3. If the nlist entity is of type logical, the corresponding clist constant must be of type logical, or hexadecimal.

4. If the nlist entity is of type character, the corresponding clist constant must be of type character or hexadecimal. If the constant is of type hexadecimal, it must exactly fill the character variable, array element, or substring.

A noncharacter entity becomes defined with a hexadecimal value according to the rules for input using Z format.

If necessary, the clist constant is converted to the type of the nlist entity according to the rules for input. The type of conversion performed on a constant appearing in an initial value list (clist) when it defines the initial value of a variable in a list (nlist) in a Data statement is explained as follows:

Conversion during Data initialization

CONSTANT TYPE	VARIABLE TYPE					
	INTEGER	REAL	DOUBLE PRECISION	COMPLEX	LOGICAL	CHARACTER
INTEGER	n	real	dble	cmplx	i	i
REAL	int	n	dble	cmplx	i	i
DOUBLE PRECISION	int	real	n	cmplx	i	i
COMPLEX	i	i	i	n	i	i
LOGICAL	i	i	i	i	n	i
CHARACTER	i	i	i	i	i	n
HEXADECIMAL	n	n	n	n	n	n

n - No conversion except as previously noted for strings or hexadecimal constants.
int - Truncate to integer if magnitude does not exceed maximum integer size, integer overflow error given otherwise.
real - Convert to real number.
dble - Convert to double precision value.
cmplx - Convert to real and supply zero imaginary part.
i - Invalid combination resulting in syntax error.

An element must not be initially defined more than once in an executable program. If two entities are associated, only one may be initially defined in a DATA statement in the same executable program. Subscript expressions for array elements must be integer constant expressions except for implied-DO-variables, and substring expressions must be integer constant expressions.

IMPLIED-DO IN A DATA STATEMENT

The format of an implied-DO list in a DATA statement is:

```
( dlist, i = ml, m2 [,m3 ])
```

where dlist is a list of array element names and implied-DO lists
i is the name of an integer variable, called the implied-DO-variable, and
ml, m2, m3 are each an integer constant expression, except that the expression may contain implied-DO-variables of other implied-DO lists that have this imlied-DO list within their ranges.

The range of an implied-DO list is the list items preceding the implied DO variable. An iteration count and the values of the implied-DO- variable are established from ml, m2, and m3 exactly as for a DO-loop except that the iteration count must be positive. When an implied-DO list appears in a DATA statement, the list items are specified once for each iteration of the implied-DO list with the appropriate substitution of values for any occurrence of the implied-DO-variable. The appearance of an implied-DO-variable name in a DATA statement does not affect the definition status of a variable of the same name in the same program unit.

Each subscript expression in the item list must be an integer constant expression, except that the expression may contain implied-DO-variables of implied-DO lists that have the subscript expression within their ranges.

If the length of a character entity in the item list is greater than the length of its corresponding character constant, the additional rightmost characters in the entity are initially defined with blank characters. If the length is less than the length of its corresponding character constant, the additional rightmost characters in the constant are ignored. Note that each character constant initially defines exactly one variable, array element, or substring.

The following program excerpt:

```
LOGICAL L1, L2
DIMENSION A(6)
CHARACTER ACHAR*6, BCHAR*6, CCHAR*2
COMPLEX ZCMP
DATA INT, (A(I),I=4,6)/3.6, 2*5.2, 2.2/,ZCMP/(2.9,1.1)/
DATA L1, L2/2*.TRUE./,ACHAR, BCHAR,CCHAR/3*"123"/
```

causes the following assignments:

```
L1=.TRUE.
L2=.TRUE.
INT=3
A(4)=5.2
A(5)=5.2
A(6)=2.2
ZCMP=(2.9,1.1)
ACHAR="123   "
BCHAR="123   "
CCHAR="12"
```

SECTION 6

ASSIGNMENT STATEMENTS

INTRODUCTION

The executable assignment statements allow an arithmetic, logical or character expression, or a statement label to be assigned to a variable.

ARITHMETIC ASSIGNMENT STATEMENT

The arithmetic assignment statement permits real, double precision, complex, and integer type variables to be assigned a value.

The format of an arithmetic assignment statement is:

```
v = e
```

where v is a variable of type integer, real, double precision, or complex; and
e is an arithmetic expression.

When the arithmetic assignment statement is executed, the arithmetic expression is evaluated, and the value obtained is placed into the variable on the left of the equal sign.

In general, the variable and the arithmetic expression need not be of the same type. If the types are different, the expression is first evaluated and then converted to the type of the variable. This conversion proceeds according to the rules indicated as follows:

Arithmetic Conversion and Assignment of e to v

Type of v	Value Assigned
Integer	INT(e)
Real	REAL(e)
Double Precision	DBLE(e)
Complex	CMPLX(e)

The functions in the "Value Assigned" column of the above table are generic functions as described in section 12.

The following are arithmetic assignment statements:

```
D = E(3) - 2.5
F(2)= FCN(2,3) + 3.5*I
```

LOGICAL ASSIGNMENT STATEMENT

The logical assignment statement is used to assign a value to a logical variable.

The format of a logical assignment statement is:

```
v = e
```

where v is a logical variable, and
e is a logical expression.

When a logical assignment statement is executed, the logical expression is evaluated and the resultant logical value is assigned to the variable to the left of the equal sign.

The following are examples of logical assignment statements where L1, L2, and L3 are logical variables, and A, B, and C are real variables:

```
L1 = L1.AND.L2
L3 = A.GT.B.OR.B.LT.C
L2 = .TRUE.
```

STATEMENT LABEL ASSIGNMENT-ASSIGN STATEMENT

The ASSIGN statement is used in conjunction with the assigned GO TO statement.

The format of the ASSIGN statement is:

```
ASSIGN s TO i
```

where s is a statement label, and
i is a an integer variable.

The execution of the ASSIGN statement causes a statement label to be assigned to a variable. The label must be that of an executable statement or a format statement that appears in the same program unit as the ASSIGN statement. The ASSIGN statement is the only way that a variable may be defined with a statement label value. The variable, once defined, may be referenced in an assigned GO TO statement or as a format identifier in an input/output statement. Until it has been reassigned with an arithmetic value, an assigned variable must not be referenced in any other manner. It may be redefined with a statement label or an arithmetic value at any time.

Examples of the ASSIGN statement follow:

```
ASSIGN 99 TO LA
GO TO LA, (1,2,99)

ASSIGN 10 TO FRMT
WRITE (5,FRMT) I
10 FORMAT (I3)
```

CHARACTER ASSIGNMENT

The character assignment statement is used to assign a value to a character variable or character substring.

The format of the character assignment statement is:

```
v = e
```

where v is the name of a character variable, a character array element, or character substring, and
e is a character expression.

The execution of a character assignment statement causes the evaluation of the character expression in the statement, and the resultant value is assigned to the character variable, character array element or substring to the left of the equal sign. None of the character positions being defined in the variable or substring may be referenced in the expression. If v and e are of different length then the following rules are applied. If the length of the entity being defined is greater than the length of the expression, the entity is assigned the value of the expression with left justification and the remainder of the entity is filled with blank characters. If the length of the variable is less than the length of the expression, the expression is truncated from the right prior to assignment to the variable.

The following are examples of character assignment statements:

```
CHARACTER C1(5)*5,C2*2,C3*8
C3=C1(1)//C2//"X"
C1(1)(2:3)="A"//C2
C2 = C3(1:2)
```

SECTION 7

CONTROL STATEMENTS

INTRODUCTION

The executable control statements are used to alter the normal flow of program execution. These statements may transfer control to another part of the program, terminate or suspend execution, or control iterative processes.

The control statements consist of the following:

Unconditional GO TO
Assigned GO TO
Computed GO TO
Arithmetic IF
Logical IF
Block IF, ELSE IF, ELSE, END IF
DO
CONTINUE
PAUSE
STOP
END
CALL (See section 11.)
RETURN (See section 11.)

GO TO STATEMENTS

The executable GO TO statements transfer control from one point of an executing program to another point in the same program unit.

UNCONDITIONAL GO TO

The format for an unconditional GO TO statement is:

```
GO TO s
```

where s is the label of an executable statement in the same program unit.

For example, the statement GO TO 23 causes the statement labeled 23 to be the next statement executed.

ASSIGNED GO TO

The format for an assigned GO TO statement is:

```
GO TO i [[,](s[,s]...)]
```

where i is an integer variable, and
s is the label of an executable statement that appears in the same program unit.

The execution of this statement causes control to be transferred to the statement whose label was last assigned to the variable by an ASSIGN statement in the same program unit as the assigned GO TO statement. If the parentheses list is present, the statement label assigned to the variable must be one of the statements in the list.

Examples follow:

```
ASSIGN 9 TO IJK
GO TO IJK, (9,1,2)
```

Control would be transferred to the statement labeled 9.

```
ASSIGN 2 TO KI
GO TO KI
```

Control would be transferred to the statement labeled 2.

COMPUTED GO TO

The format for a computed GO TO statement is:

```
GO TO (s[,s]...) [,]i
```

where i is an integer expression, and
s is the label of an executable statement that appears in the same program unit as the computed GO TO statement.

Execution of this statement causes control to be transferred to a statement whose label appears in the list or to the next executable statement following the GO TO statement. How control is transferred depends on the value of the expression following the list. It is evaluated and is used to select one of the labels in the list. If the expression has the value n, control passes to the n-th label in the list. If there are fewer than n labels in the list or if n is less than or equal to zero, control passes on to the next executable statement following the GO TO statement.

An example of a computed GO TO statement is:

```
GO TO (11,25,3,9,11,17), J+1
```

At execution time, the value of J+1 is computed. If J+1=4, then control will pass to the statement labeled 9, the fourth label in the list. If J+1 is 1 or 5 in this example, control will pass to statement number 11. If J+1 is less than 1 or greater than 6, control will pass to the next executable statement after the GO TO statement.

IF STATEMENTS

ARITHMETIC IF

The arithmetic IF statement is a three-way branch.

The format for the arithmetic IF is:

```
IF (e) S1,S2,S3
```

where e is an integer, real, or double precision expression.
S1,S2,S3 are labels of executable statements in the same program unit.

The same statement label may appear more than once in the IF statement. Execution of the arithmetic IF statement causes the arithmetic expression within the parentheses to be evaluated. Control is transferred to the statement bearing the first, second, or third label, depending on whether the value of the expression is less than zero, zero, or greater than zero, respectively.

Examples of arithmetic IF statements follow:

```
IF (I-J) 5,10,15
```

If I-J is negative, control will be transferred to the statement labeled 5; if zero, to the statement labeled 10; or if positive, to the statement labeled 15.

```
IF (J*5-B+2) 3,2,3
```

In this example control passes to the statement labeled 2 only if the expression within the parentheses is zero. Otherwise, control passes to the statement labeled 3.

LOGICAL IF

The logical IF statement conditionally executes a statement.

The format of the logical IF statement is:

```
IF (e) s
```

where e is a logical expression, and
s is any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END or another logical IF statement.

Execution of the logical IF causes evaluation of the logical expression within parentheses. If the logical expression is true, then the statement following the logical expression is executed. If the logical expression is false, then the statement following the logical expression is ignored and control passes on to the next executable statement following the IF statement.

Examples of logical IF statements:

```
IF (A.EQ.B) G=G+1
```

If A equals B then G will be incremented by 1. Otherwise, G remains unchanged. In any event, control then passes on to the next statement.

```
IF (L1) GO TO 97
```

If L1 is true, then control will pass to statement number 97. If L1 is false, then control will pass on to the next statement.

```
IF (A.LE.97) IF (B) 12,12,13
```

If A is less than or equal to 97, then the Arithmetic IF will be executed, and control will pass on to statement number 12 or 13, depending on the value of B. If A is greater than 97, control will pass on to the next statement.

BLOCK IF, ELSE IF, ELSE, END IF STATEMENTS

These statements facilitate the conditional execution of a statement or group of statements.

Block IF Statement

The block IF statement is used with the END IF statement and optionally the ELSE IF or ELSE statements to control execution sequence. The block IF may have associated an ELSE or an ELSE IF statement. Nevertheless, it must have a corresponding END IF statement.

The format of the block IF statement is:

    IF (e) THEN

where e is a logical expression.

The IF-level of a statement is determined by the following calculation:

    IF-level = X1-X2

where X1 is the number of block IF statements from the beginning of the program unit up to and including this statement, and
X2 is the number of END IF statements in the program unit up to, but not including, this statement.

Every statement must have an IF-level greater than or equal to zero. Each block IF, ELSE, ELSE IF, and END IF must have an IF-level greater than zero.

All the statements that appear following the block IF statement up to, but not including the next END IF, ELSE or ELSE IF that has the same IF- level as the block IF are called the IF block. An IF block may be empty. Execution of the block IF statement causes evaluation of the logical expression contained in the statement. If the value of the expression is true, normal execution sequence continues to the statements within the IF block. If the IF block is empty, control is transferred to the next END IF statement with the same IF-level as the block IF statement. When the value of the expression is false, control is transferred to the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement.

Transfer of control into an IF block from outside the IF block is not permitted. If execution of the last statement in the IF block does not result in a transfer of control, control is transferred to the next END IF statement that has the same IF-level as the block IF statement that precedes this IF block.

Block IF statements may be nested within IF blocks, ELSE blocks, or ELSE IF blocks. The entire series of statements from the block IF through the END IF must be contained within the block.

ELSE IF Statement

This statement is used optionally as the else part associated with a block IF statement. It must have corresponding block IF and END IF statements, and may optionally have an else part of either an ELSE statement or another ELSE IF statement.

The format of the ELSE IF statement is:

    ELSE IF (e) THEN

where e is a logical expression.

All of the statements appearing following the ELSE IF statement up to, but not including, the next ELSE, ELSE IF or END IF that has the same IF level as the ELSE IF are called the ELSE IF block.

Execution of the ELSE IF statement causes evaluation of the logical expression in this statement. If this value is true, execution continues to the first statement of the ELSE IF block. If the value is true and the ELSE IF block is empty, control is passed to the next END IF that has the same IF-level as the ELSE IF statement. If the value of the expression in the ELSE IF statement is false, control is transferred to the next ELSE IF, ELSE, or END IF that has the same If-level as the ELSE IF statement.

If no statement within the ELSE IF block causes a transfer of control outside of the ELSE IF block, control is transferred to the next END IF statement that has the same IF-level as the ELSE IF statement that precedes this ELSE IF block.

Transfer of control into an ELSE IF block is not permitted.

ELSE Statement

This statement is used in conjunction with the block IF or ELSE IF statements.

The format of the ELSE statement is:

```
ELSE
```

All the statements following the ELSE statement up to, but not including the next END IF statement with the same IF-level as the ELSE statement are called an ELSE-block. An END IF statement of the same If-level as the ELSE statement must appear before the appearance of an ELSE IF or ELSE statement of the same If-level.

The ELSE statement functions like a CONTINUE statement; however, control may not be transferred to a ELSE statement or into an ELSE block.

END IF Statement

The END IF statement functions like a CONTINUE statement. For each block IF statement there must be a corresponding END IF statement in the same program unit. The corresponding END IF statement is the next END IF statement that has the same IF-level as the block IF statement.

The following examples of the block IF, ELSE IF, ELSE and END IF statements are in order of increasing complexity.

```
J=0
I=0
IF ( L1) THEN
I=1
J=2
END IF
```

After the above statements have been executed, if L1 is true, I=1 and J=2. If L1 is false, I=0, and J=0.

```
IF ( L2 ) THEN
I=1
ELSE
I=2
END IF
```

In the above example, if L2 is true I is set equal to 1. If L2 is false I is set equal to 2.

```
X=0.0
Y=1.0
Z=5.0
IF ( X.EQ.O. ) THEN
Y=3.3
ELSE IF ( Z.GT.O. ) THEN
Y=2.1
END IF
```

After the above statements have been executed, X=0.0,Y=3.3, and Z=5.0.

DO STATEMENT

The executable DO statement is a control statement that allows a series of statements to be repeatedly executed while the value of a control variable is varied between specified limits. The number of iterations of these statements is dependent upon this control variable. The group of statements is referred to as a DO loop.

The format for the DO statement is:

```
DO s[,] i = e1,e2 [,e3]
```

where
- s is the label of an executable statement following the DO statement within the same program unit. This statement s is called the terminal statement of the DO-loop.
- i is an integer, real, or double precision variable called the DO-variable.
- e1,e2,e3 are the initial, terminal, and incremental parameters, respectively, each of which is an integer, real, or double precision expression. If the incremental parameter is omitted, the value of 1 is assumed.

The DO statement causes repeated execution of the statements in its range. The range of a DO statement consists of all the executable statements following the DO statement up to and including the terminal statement specified in the DO statement.

The terminal statement of a DO-loop must not be an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement. If the terminal statement of a DO-loop is a logical IF statement, it may contain any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement.

Any number of DO statements may be nested one within another with the following restriction: if a DO statement occurs in the range of another DO, the range of the former must be completely contained within the range of the latter. Both may, however, specify the same statement as the terminal statement in their ranges.

If a DO statement appears within an IF-block, ELSE IF-block, or ELSE- block, the range of that DO-loop must be contained entirely within that IF-block, ELSE IF-block, or ELSE-block, respectively. Also, if a block IF statement appears within the range of a DO-loop, the corresponding END IF statement must appear within the range of that DO-loop.

Transfer of control into the range of a DO-loop is not permitted. A DO-loop is either inactive or active. A DO-loop is initially inactive and becomes active only upon execution of the DO statement. Once active it becomes inactive when any of the following occur:

1. Its iteration count is found to be equal to zero.

2. A RETURN, STOP or END is executed within its range.

3. Control is transferred to a statement outside of the DO loop range.

When a DO-loop becomes inactive the DO-variable retains its last value. Execution of a DO-loop causes the following processing phases to take place:

1. Loop Initialization Processing
   The initial terminal and incremental parameters of the DO-loop are established by evaluating the respective expressions in the DO statement. The DO-variable is assigned the value of the initial parameter, and the iteration count is established. The iteration count is calculated by the following expression where M1, M2, and M3 are the initial, terminal, and incremental parameters: MAX (INT((M2-M1+M3)/M3),0) The iteration count is zero if M1 > M2 and M3 > 0 or M1 < M2 and M3 < 0.

2. Loop Control Processing
   This processing determines if further execution of the range should take place. The iteration count is tested. If the iteration count is greater than zero the range of the loop is executed. However, if the iteration count is zero, the DO-loop becomes inactive and the range of the DO is not executed. Execution continues with the first executable statement following the terminal statement of the DO statement. However, if some of the DO-loops sharing the terminal statement are active, execution continues with the incrementation processing.

3. Execution of the Range
   The statements within the range of the DO-loop are executed until the terminal statement is reached. Incrementation processing follows. The DO-variable of the DO statement may not be assigned a new value while the DO-loop is active.

4. Terminal Statement Execution
   Execution of the terminal statement occurs as a result of the normal execution sequence or as a result of transfer of control. Unless execution of the terminal statement results in a transfer of control, execution then continues with incrementation processing.

5. Incrementation Processing
   Loop incrementation for the active DO-loop whose DO statement was most recently executed consists of two steps. The DO-variable is incremented by the value of the incremental parameter, and the iteration count is decremented by one. Following these two steps execution continues with loop control processing of the DO-loop whose iteration count was decremented.

It should be noted that because the iteration count test is done before executing the range of the DO-loop the range of the DO-loop may not be executed at all.

Consider the following examples:

```
    N=0
    DO 100 I=1,10
    J=I
    DO 100 K=1,5
    L=K
100 N=N+1
101 CONTINUE
```

After execution of these statements and at the execution of the CONTINUE statement, I=11, J=10, K=6, L=5, and N=50.

```
    N=0
    DO 200 I=1,10
    J=I
    DO 200 K=5,1
    L=K
200 N=N+1
201 CONTINUE
```

After execution of these statements and at the execution of the CONTINUE statement, I=11, J=10, K=5, and N=0. L is not defined by these statements.

In the following example Z is calculated as the sum of the cosines of -5.4, -5.3, -5.2, and 5.1:

```
   Z=0
   DO 10 X=-5.4,-5.05,.1
10 Z = COS(X) + Z
```

The following example initiates every other element of the arrays A and B, A(1) through A(9) and B(1,1) through B(9,9):

```
       IJ=10
       DO 5 I = 1, IJ, 2
       A(I)= 5.
       DO 5 J = 1, IJ, 2
     5 B(I,J)= 5.
```

CONTINUE STATEMENT

The executable CONTINUE statement produces no action. It is frequently used as the terminal statement of a DO-loop.

The format for the CONTINUE statement is:

```
     CONTINUE
```

Following is an example:

```
         .
         .
         .
       DO 1 I=2,10,2
       A(I)=I/M
       IF (A(I)) 3,1,1
     1 CONTINUE
       M=-M
     3 CONTINUE
         .
         .
         .
```

In this example, the CONTINUE statement labeled 1 is used as the final statement of a DO loop, and the CONTINUE statement labeled 3 is used as a transfer point for an arithmetic IF statement. The use of the first CONTINUE allows the DO loop to be clearly delimited. The second CONTINUE statement allows transfer to an arbitrary point in the program, the location of which may be changed by merely changing the location of the CONTINUE statement.

PAUSE STATEMENT

The executable PAUSE statement allows an executing program to be suspended indefinitely.

The format for the PAUSE statement is:

```
     PAUSE [n]
```

where n is a string of one to five digits or a character constant.

The execution of the PAUSE statement causes the unconditional suspension of the program being executed pending operator action. In addition to suspending the program, the execution of this statement causes the optional digits or character constant following the PAUSE to be displayed at the Operator Display Terminal.

The program may be resumed at the first executable statement following the PAUSE statement by a system input message from the Operator Display Terminal of the form:

    mOK

where m is the mix number of the task. Alternately, the task may be discontinued at this point by a system input message of the form:

    mDS

where m is the mix number of the task. If the program is run with CANDE the ?OK and ?DS CANDE control commands may be used for the mOK and nDS system input messages, respectively.

Examples follow:

```
PAUSE
PAUSE 2
PAUSE 'BEFORE PUNCHING'
PAUSE "BEFORE PUNCHING"
```

STOP STATEMENT

The executable STOP statement allows termination of an executing program.

The format for the STOP statement is:

    STOP [n]

where n is a string of one to five digits or a character constant.

The execution of the STOP statement causes termination of the program. In addition, the optional digits or character constant following the STOP to be displayed at the Operator Display Terminal.

Examples follow:

```
STOP
STOP 6
STOP "PHASE 1"
```

END STATEMENT

The END statement indicates the end of a program unit.. When it is executed in a function or subroutine subprogram, it has the effect of a RETURN statement. When it is executed in a main program, it terminates the execution of the program.

The format of an END statement is:

```
END
```

The END statement must not be continued and must be an initial line. No other statement may have an initial line that appears to be an END. The END statement must be the last line of every program unit.

SECTION 8

INPUT/OUTPUT STATEMENTS

INTRODUCTION

Input statements provide a method for transferring data from peripheral hardware devices (such as disk pack, magnetic tapes and remote terminals) to internal storage or from an internal file (type CHARACTER datum) to internal storage. This process is called reading.

Output statements provide the means of transferring data from internal storage to external media or from internal storage to an internal file. This process is referred to as writing. Some input/output (I/O) statements allow for editing of the data at the time of the transfer.

In addition to the statements that transfer data, there are auxiliary I/O statements which permit the positioning of peripheral devices and allow for the properties of the external media to be specified or interrogated.

The following are data transfer I/O statements:

```
READ
WRITE
PRINT
PUNCH
```

The following are auxiliary I/O statements:

```
OPEN
CLOSE
INQUIRE
BACKSPACE
ENDFILE
REWIND
FIND
FILE
```

FILES

A FORTRAN program has access to a file which is a collection of records, via input/output statements. Records may contain a sequence of values or a sequence of characters. A record that is transferred with a formatted input/output statement is called a formatted record. It is composed of a sequence of characters. A record that is transferred unformatted, without editing specifications is called an unformatted record. An unformatted record is a sequence of values in their internal machine representation. The entities transferred are transferred from internal storage unchanged.

In addition to formatted and unformatted data records there is an ENDFILE record. An ENDFILE record is written by an ENDFILE statement. It may occur only as the last record of a file, and does not have a length property.

The N records contained in a file are logically numbered in order from 1 through N; record 1 immediately follows the initial point of the file, while record N immediately precedes the endfile mark terminating the file.

There are two kinds of files, external and internal. External files are used to transfer data to and from peripheral devices. Internal files provide a means of transferring data from internal storage to internal storage.

An internal file has the following properties:

1. The file is a character variable, character array element, character array, or character substring.

2. A record of an internal file is a character variable, character array element, or character substring.

3. If the file is a character variable, character array element, or character substring, it consists of a single record whose length is the same as the length of the variable, array element, or substring respectively. If the file is a character array, it is treated as a sequence of character array elements. Each array element is a record of the file. The ordering of the records of the file is the same as the ordering of the array elements in the array. Every record of the file has the same length, which is the length of an array element in the array.

4. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is filled with blanks.

5. A variable, array element, or substring that is a record of an internal file may also be used in a different context in the same FORTRAN77 program, e.g., in a character assignment statement.

6. An internal file is always positioned at the beginning of the first record prior to data transfer.

At any given time, there is a set of files that are said to exist for a FORTRAN 77 program. A file may be present on a system, yet not exist for a FORTRAN 77 program at a particular time. For example, security reasons may prevent a file from existing for a given program. A file may exist and contain no records; an example of this is a newly created file into which records have not been written. To create a file means to cause a file to exist that did not previously exist. To delete a file means to terminate the existence of the file. All input/output statements may refer to files that exist. INQUIRE, OPEN, CLOSE, WRITE, PRINT, and PUNCH statements may also refer to files that do not exist.

Fortran provides two methods of accessing the data records of an external file: sequential and direct (commonly called random). When a file is accessed sequentially the records are processed in sequence, from the first in the file through the last. The direct access method allows for the access of records of a file in any order. The records are accessed via their record numbers relative to their position in the file. Note that internal files must only be accessed sequentially.

UNITS

A unit is a means of referring to a file. All unit numbers explicitly appearing in FILE statements or in input/output statements (except INQUIRY) are said to exist for a program. Unit numbers 5,6, and 7 also always exist for a program regardless of being referenced explicitly. In order to process an external file it is necessary that a file be connected to a input/output unit. Unit numbers 5, 6, and 7 and unit numbers appearing in FILE statements are preconnected.

Each unit identifier has a default device type associated with it. Units 5,6, and 7 are assigned card reader, line printer, and card punch files, respectively, by default. Other designators are assigned disk files by default.

A file may be connected and not exist. An example is a preconnected new file.

A unit must not be connected to more than one file at the same time, and a file must not be connected to more than one unit at the same time. However, means are provided to change the status of a unit and to connect a unit to a different file. After a unit and a file have been disconnected by the execution of a CLOSE statement, a unit may be connected again within the same executable program to the same file or a different file.

Each file is recognized by the system by two names: the internal name and the external name or the value of the TITLE attribute of the file (see Input/Output Subsystem, Form No. 5001779). The form of an internal file name is:

FILEn

where n is the unit identifier for the file with no leading zeros. The highest unit number is 99.

The internal file names corresponding to the units 6 and 99 are FILE6 and FILE99, respectively.

The TITLE of a file consists of one to fourteen identifiers separated by slashes (/); each identifier is one to seventeen characters long. When the TITLE of a program file is unspecified, a default TITLE is assigned to the file which is identical to the internal file name.

DATA TRANSFER INPUT/OUTPUT STATEMENTS

The basic components of data transfer I/O statements are the control information list and the input/output list.

CONTROL INFORMATION LIST

The control information list may contain at most one of each of the following items, separated by commas:

```
[UNIT =] u
[FMT=]f
[REC=]r
IOSTAT=i
ERR=s
END=s
```

Following are some restrictions on the items in the control information list:

1. The list must contain exactly one unit specifier.

2. If the list contains a record specifier, it must not contain an end of file specifier nor a format specifier with a format identifier of asterisk.

3. A write statement must not have an end of file specifier in its list.

4. If the unit specifier specifies an internal file, the list must contain a format identifier other than an asterisk and must not contain a record specifier.

Below is detailed information on control list items. Examples may be found following the discussion of the READ, WRITE, PRINT, and PUNCH statements.

Unit Specifier

The unit specifier designates the external unit to be used for the input/output statement in which it appears.

The format of a unit specifier is:

```
[UNIT =] u
```

where u is an external unit identifier or an internal file identifier.

If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in a list of specifiers of an input/output statement.

An external unit identifier is used to refer to an external file. It may be one of the following:

1. An integer expression whose value must be greater than or equal to zero and less than 100.

2. An asterisk, identifying units 5, and 6 that are preconnected for formatted sequential read, and write, respectively.

In auxiliary input/output statements the asterisk form may not be used.

The external unit identified by a value is the same external unit in all program units of the executable program, as shown in the following example:

```
SUBROUTINE A
READ (6) X
.
.
SUBROUTINE B
N=6
REWIND N
```

The value 6 used in both program units identifies the same external unit.

An internal file identifier is used to refer to an internal file. It can be the name of a character variable, character array, character array element, or character substring.

Format Specifier

The format specifier designates the format to be used for the I/O statement in which it appears.

The format of a format specifier is:

```
[FMT =] f
```

where f is a format identifier.

A format identifier identifies a format. It must be one of the following:

1. The statement label of a FORMAT statement that appears in the same program unit as the format identifier.

2. A integer variable name that has been assigned the statement label of a FORMAT statement that appears in the same program unit as the format identifier.

3. A character array name.

4. Any character expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant. Note that a character constant is permitted.

5. An asterisk, specifying list-directed formatting.

If the optional characters FMT= are omitted from the format specifier, the format specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters UNIT=.

Record Specifier

The record specifier is used to specify the number of the record that is to be read or written with the I/O statement in which it appears.

The format of a record specifier is:

    [REC =] r

where r is an integer expression whose value is positive. This is the record number.

If the optional characters REC= are omitted from the record specifier, the unit specifier must appear without the optional characters UNIT=, and the unit specifier and record specifier must be combined in the form:

    u'r

to form one list item, which must be the first item in the control information list.

The record number must have a value greater than or equal to 1 and less than or equal to the number of records in the file for which the I/O will be done.

Input/Output Status Specifier

The I/O status specifier provides information regarding the result of the I/O operation.

The format of the input/output specifier is:

IOSTAT=ios

where ios is a integer variable or an integer array element.

Execution of an input/output statement containing this specifier causes the specifier to become defined:

1. With a zero value if neither an error condition nor an end-of-file condition is encountered.

2. With a positive integer value if an error conditon is encountered. Ios is interpreted by determining the class and subclass of the integer value where:
   class = ios/1000 (integer division)
   subclass = MOD(ios,1000)
   Class designates a general error condition and subclass a refinement of that general error condition. The value of subclass only has meaning within a particular class. If a class has no associated subclass (that is, MOD(ios,1000) .EQ. 0) then there is no additional information availabel concerning the nature of the error.

Valid values for class, and their associated subclasses are:

Class	Subclass	Description
1000		I/O Status
	00	Unknown error
	10	Security error
	30	Parity error
	100	Invalid record number
	110	Break on datacom output
	120	I/O timeout
	130	Format error
	140	Data error
	150	Record length error
2000		Open error
	00	Unknown error
	10	Security error
	30	Parity error
	40	File already open
	100	File exists but is not available
	110	File does not exist
	120	Geneology mismatch
	130	Serial number mismatch
	140	Non-resident backup file
	150	No resource available
	160	Pack not mounted
3000		Close error
	00	Unknown error
	40	File already closed
	100	Record count error
	110	Block count error

3. With a negative integer value if an end-of-file condition is encountered and no error condition is encountered.

If an IO error or end-of-file condition occurs and no error or end-of-file specifier is present, the presence of an IO status specifier prevents termination of the program. Execution continues with the next executable statement.

Error Specifier

The error specifier provides the ability to alter the execution sequence when an error condition occurs during the execution of and I/O statement.

The format of an error specifier is:

```
ERR=s
```

where s is the statement label of an executable statement that appears in the same program unit as the error specifier.

If an I/O statement contains an error specifier and the processor encounters an error condition during execution of the statement execution of the I/O statement terminates, if the I/O statement contains an I/O status specifier, the I/O specifier becomes defined with a value corresponding to the error condition, and execution continues with the statement label s.

End-Of-File Specifier

The end-of-file specifier provides the ability to alter the execution sequence when an end-of-file condition is encountered during an I/O statement.

The format of an end-of-file specifier is:

```
END=s
```

where s is the statement label of an executable statement that appears in the same program unit as the end-of-file specifier.

If the end-of-file specifier appears in a READ or WRITE statement, then control will automatically pass to the statement labeled s when an end-of-file condition occurs if no error condition has occurred. End-of-file condition occurs when any of the following actions are attempted by a READ or WRITE statement.

Statement	Type of File	Action Causing End-of-File Condition
READ	CARD	Attempting to read a card with an invalid character in column 1.
READ	TAPE	Attempting to read an endfile record.
READ	DISK	Attempting to read a record beyond the last record of a file.
WRITE	DISK	Attempting to write beyond the end of the designated area of the disk file.
WRITE	LINE PRINTER	Attempting to write a record at end-of-page. The PAGESIZE file attribute must be assigned an appropriate value on a FILE system control card or file declaration.

If the conditions occur and the end-of-file specifier is missing, then execution of the program will be terminated unless an IOSTAT specifier is present.

INPUT/OUTPUT LIST

The I/O list specifies the data entities to which or from which data is to be transferred.

An I/O list is a list of I/O list items and implied-DO lists. An input list item may be a variable name, an array element name, an array name, or a character substring name.

An output list item may be a variable name, an array element name, an array name, a character substring name or an an expression; however, a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses is not permitted unless the operand is the symbolic name of a constant.

Note that a constant, an expression involving operators or function references, or an expression enclosed in parentheses may appear as an output list item but must not appear as an input list item.

Implied-DO List

The format for an implied-DO list is:

```
(dlist, i=e1,e2 [,e3])
```

where dlist is an I/O list.

i is an integer, real, or double precision variable and

e1,e2,e3 are each integer, real, or double precision expressions

The range of an implied-DO list is the list dlist. Note that the list may contain implied-DO lists. The iteration count and the values of the DO-variable are established from e1, e2, and e3 exactly as for a DO-loop. In an input statement, the DO-variable must not appear as an input list item.

Transfer of data to and from list elements occurs in the order they are named in the I/O list, from left to right. Items named in an implied-DO I/O list are referenced repeatedly until the implied-DO is satisfied. The appearance in an I/O list of an array element specifies only that array element, while the appearance of an array name specifies every element declared for the array in that program unit. These array elements are transferred in the order in which they are stored internally.

Following are examples of I/O lists which may be used for input or output:

```
A,B(3)
XARRAY, XARRAY (2,4)
(A(J) , B , J , C(J,J) , J = 1,10,2)
(IPLOT , COUNT = 1,60)
(X, (Y(I,J), I=1,10), Z, J=1,2)
XARRAY(3), XARRAY, (XARRAY(EL), EL = 1,20)
RESULT
(A(I), I=1,20) , (J(I), I=1,5)
(((Z(I,J,K), K=1,10), J=1,10),I=1,10),A,G
```

Follow are examples of I/O lists which may be used for output only:

```
1+COS(N) , X(1+N), G
A .OR. B
A, (J,GX(J), B, J=3,4),10,X
PARTA // PARTZ
```

READ, WRITE, PRINT AND PUNCH STATEMENTS

The READ statement is a data transfer input statement. The WRITE, PRINT, and PUNCH are data transfer output statements.

The formats for the data transfer I/O statements are:

```
READ (clist) [iolist]
READ f [,iolist]
WRITE (clist)[iolist]
WRITE f [,iolist]
PRINT f [,iolist]
PUNCH f [,iolist]
```

where clist is a control information list.
f is a format identifier.
iolist is an I/O list.

A data transfer statement of form 1 or 3 contains a control information list which must include a unit specifier that identifies an external unit or an internal file.

Forms 2, 4, 5, and 6 which do not contain control information lists specify units 5, 6, and 7. The READ statement form 2 specifies unit 5. The WRITE statement (form 4) and the PRINT statement specify unit 6, and the PUNCH statement specifies unit 7.

The following are examples of the READ, WRITE, PRINT and PUNCH statements. In these examples 10, 95, and 99 are statement labels; 99 is used as the statement label for error specifiers; 10 is the label on a format statement; and 95 is used as the statement label for end-of-file specifiers. A 9 appearing in the following examples is a unit specifier.

```
READ  (9,10,END=95) A
READ  (ERR=99,UNIT=9,REC=3,FMT=10) A,B
WRITE  (9) A,B
WRITE  (9'3,10,ERR=99) A
WRITE  (UNIT=9,FMT= '(5X,I6,I9)') A,B
PRINT 10
PUNCH 10, A, B
```

Following is an example of a read from the default unit 5.

```
READ (*,10,ERR=99) A
```

Following are two examples of list-directed write statements to the default unit 6.

```
WRITE (*,*) A,B
WRITE (IOSTAT=RESLT,UNIT=*,FMT=*) A,B
```

Following are examples of data transfer I/O statements from an internal file (type character datum) to internal storage (READ), and from internal storage (WRITE) to an internal file.

```
DIMENSION IARY (10)
CHARACTER CRD*80
READ (CRD,FMT=10) (IARY (I) ,I=1,4)
10 FORMAT  (2I6,56X,2I6)
```

In the above example the first twelve characters and the last twelve characters of the variable CRD are transferred as four integer values into the four elements of the array IARY. One record is read.

```
   DIMENSION A (6)
   CHARACTER X (6) *11
   WRITE (X,FMT=15) A
15 FORMAT (I5)
```

In the preceding example six internal file records are written. The six elements of the array A are transferred into the six elements of the array X. Each five character value will be left justified in each element and the remaining two characters of each element will be blanks.

AUXILIARY INPUT/OUTPUT STATEMENTS

OPEN STATEMENT

An OPEN statement may be used to:

1. Connect an existing file to a unit.
2. Create a file that is predefined as connected to a unit.
3. Create a file and connect it to a unit.
4. Change certain attributes of the connection between a file and a unit.

The format of an OPEN statement is:

```
OPEN(oplist)
```

where oplist is a list of specifiers.

The allowable specifiers for the OPEN statement are the following:

```
fa = v
[UNIT = ] u
ERR = s
IOSTAT = i
FILE = n
STATUS = sta
ACCESS = acc
FORM = fm
RECL = l
BLANK = bl
```

The list in the OPEN statement must contain a unit specifier and may contain at most one of each of the specifiers ERR, IOSTAT, FILE, STATUS, ACCESS, FORM, RECL, and BLANK. If a file is to be connected for direct access the RECL specifier must be used.

The specifiers for the OPEN statement are described as follows:

fa = v	is a file attribute specifier. fa is a file attribute (see INPUT/OUTPUT Subsystems manual, Form no. 5001779). v is an expression whose type and value is valid for the associated file attribute. If v is an attribute mnemonic it must appear in quotes.

[UNIT =] u — is an external UNIT specifier. The definition of this specifier may be found in the control information list description.

ERR=s — is an error specifier. The definition of this specifier may be found in the control information list description.

IOSTAT=i — is an I/O status specifier. The definition of this specifier may found in the control information list description.

FILE=n — n is a character expression whose value is the external name of the file that is connected to the specified unit. The character expression must be terminated with a period.

STATUS=sta — sta is a character expression, whose value when trailing blanks are removed may be the following. If OLD or NEW is specified a FILE specifier must be given. UNKNOWN is default.

- OLD — the referenced file is expected to exist.
- NEW — a new file is created.
- SCRATCH — a temporary file is created. This file will only exist for the execution of the program, or until a CLOSE statement is executed.
- UNKNOWN — This specification is the same as SCRATCH.

ACCESS=acc — acc is a character expression, whose value when trailing blanks are removed may be the following:

- SEQUENTIAL — indicates sequential access will be performed.
- DIRECT — indicates direct access will be performed.

If this specifier is omitted sequential is assumed.

FORM=fm — fm is a character expression whose value when trailing blanks are removed may be the following:

- FORMATTED — indicates formatted I/O will be performed.
- UNFORMATTED — indicates unformatted I/O will be performed.

If this specifier is omitted, UNFORMATTED is assumed if the file has been connected for direct access and FORMATTED is assumed if the file has been connected for sequential access.

RECL=l — l is a positive integer expression. This specifier is required and must only be specified when a file is connected for direct access. It specifies the length of the records of the file. If the file is being connected for formatted I/O, the length is the number of characters. If the file is being connected for unformatted I/O the length is measured in words.

BLANK=bl

bl is a character expression whose value when trailing blanks are removed may be the following:

NULL — All blank characters in numeric formatted input fields on the specified unit are ignored except that a data field of all blanks has a value of zero.

ZERO — All blanks other than leading blanks are treated as zero.

If this specifier is omitted, a value of NULL is assumed.

If a unit is connected to a file that exists, execution of an OPEN statement for that unit is permitted. If the FILE specifier is not included in the OPEN statement, the unit remains connected to the same file.

If the file specified in an OPEN statement does not exist, but is the same as the one to which the input/output unit is preconnected, then the actions specified by the OPEN statement are performed.

If the file to be connected to the unit is not the same as the one to which the unit is connected, the effect is as if a CLOSE statement without a STATUS specifier had been executed, followed by the OPEN statement.

If the file to be connected to the input/output unit is the same as the one to which the unit is already connected, only the BLANK specifier may have a value different from the one currently in effect. The position of the file is unaffected.

Execution of the OPEN statement causes the new specifier values to become effective. Sequential files are repositioned to the beginning-of-file. If the file is connected to a unit, execution of an OPEN statement for the same file and a different unit is not permitted.

If a file is preconnected via a FILE statement and there is no explicit OPEN on that file, specifiers that do not appear in the FILE statement will have the same default values as if the file had been connected via an OPEN statement.

Examples of the OPEN statement follow. ACSTYP is a character variable.

```
OPEN (UNIT=8, FILE="A/B.", STATUS="OLD")
OPEN (3,STATUS="NEW", ACCESS=ACSTYP ,RECL=185)
OPEN (5, BLANK="ZERO")
```

CLOSE STATEMENT

The CLOSE statement is used to terminate the connection of a file to a unit.

The formats for the CLOSE statement are:

```
CLOSE u
CLOSE (clist)
```

where u is an external unit identifier, and
clist is a list of specifiers separated by commas.

The allowable specifiers for the CLOSE statement are:

```
[UNIT = ] u
ERR = s
IOSTAT = i
STATUS = sta
DISP = sta
```

The list in the CLOSE statement must contain a unit specifier and may contain at most one of each of the other specifiers.

The specifiers for the CLOSE statement are described as follows:

[UNIT =] u — is an external UNIT specifier. A description of this specifier is in the control information list definition.

ERR=s — is an error specifier. A description of this specifier is in the control information list definition.

IOSTAT=i — is an I/O status specifier. A description of this specifier is in the control information list definition.

STATUS=sta — is a status specifier. A description of this specifier is with the OPEN statement definition.

DISP=sta — The status specifier determines the disposition of the file that is connected to the specified unit. sta is a character expression whose value when trailing blanks are removed may be the following:

- KEEP — If KEEP is specified for a file that exists, it will continue to exist after execution of the CLOSE statement.
- DELETE — If DELETE is specified, the file will not exist after execution of the CLOSE statement.

If this specifier is omitted KEEP is assumed, unless a status specifier of SCRATCH has been defined, in which case DELETE is assumed.

Execution of a CLOSE statement need not occur in the same program unit as the execution of an OPEN statement for that input/output unit. Execution of a CLOSE statement for a unit that is not connected to any file is permitted and is ignored.

After a unit has been disconnected from a file by a CLOSE statement, it may be connected again within the same FORTRAN77 program, either to the same file or to a different file, and the file, if it still exists, may be connected again, either to the same unit or to a different unit. At the end of execution of a FORTRAN77 program, all existing connections between units and files are closed. The effect is the same as if a CLOSE statement without a STATUS specifier were executed on each connected unit.

Following are examples of the CLOSE statement: CHR is a character variable and IOST is an integer variable.

```
CLOSE(UNIT=8,STATUS="KEEP",IOSTAT=IOST)
CLOSE(3,STATUS="DELETE",ERR=100))
CLOSE(UNIT=9,STATUS=CHR)
```

INQUIRE STATEMENT

An INQUIRE statement may be used to interrogate the properties of a particular file or a file connected to a particular I/O unit. All value assignments are done according to the rules for assignment statements.

The INQUIRE statement may be executed before, while, or after a file is connected to a unit. Current values of properties at the time the statement is executed are returned.

The format of the INQUIRE statement for a file is:

```
INQUIRE(iflist)
```

where iflist is a list of specifiers, separated by commas that contain exactly one FILE specifier and at most one of each of the other specifiers.

The FILE specifier has the same format as for the OPEN statement. The named file is the file whose properties are being interrogated. It need not exist or be connected to a unit.

The format of the INQUIRE statement for a unit is:

```
INQUIRE(inlist)
```

where inlist is a list of specifiers, that contain exactly one external UNIT specifier and at most one of each of the other inquiry specifiers, except the file attribute specifiers.

The unit specified need not exist or be connected to a file. If it exists and is connected to a file, the inquiry is being made about the properties of that file.

The following inquiry specifiers may be used in either form of the INQUIRE statement:

fa =v	is a file attribute specifier (see INPUT/OUTPUT Subsystem manual, form 5001779). v is a variable or array element whose type corresponds to the type of that attribute. Execution of an INQUIRE statement containing a file attribute specifier causes the specified variable or array element to be assigned the value of the associated attribute.
IOSTAT=i	is an I/O status specifier. A description of this specifier is in the control information list definition.
ERR=s	is an error specifier. A description of this specifier is in the control information list definition.
EXIST=ex	ex is a logical variable or logical array element. Execution of the INQUIRE statement for a file causes the variable ex to be assigned the value true if the file exists; otherwise, ex is set to false. Execution of the INQUIRE statement for a unit causes ex to be assigned the value true if the specified unit exists; otherwise, ex is set to false.
OPENED=op	op is a logical variable or logical array element. Execution of the INQUIRE statement for a file causes the variable op to be set to true if the file is connected to a unit; otherwise, op is set to false. Execution of the INQUIRE statement for a unit causes op to be set to true if the specified unit is connected to a file; otherwise, op is set to false. For a preconnected unit that has not been closed, the value true is always returned, but this does not necessarily mean that the associated file is physically open.
NUMBER=n	n is an integer variable or integer array element. If there is a unit currently connected to the file, n is assigned the value of the external unit identifier. If there is no unit connected, n becomes undefined.
NAMED=nd	nd is a logical variable or logical array element. It will always be set to true.
NAME=fn	fn is a character variable or character array element. fn is assigned the value of the file name.
ACCESS=acc	acc is a character variable or character array element. acc is assigned SEQUENTIAL or DIRECT according to the type of access for which the file is connected.

SEQUENTIAL=seq	seq is a character variable or character array element that is always assigned the value YES.
DIRECT=dir	dir is a character variable or character array element that is always assigned the value YES.
FORM=fm	fm is a variable or array element of type CHARACTER. fm is assigned FORMATTED or UNFORMATTED, according to the value used in executing the OPEN statement.
FORMATTED=fmt	fmt is a character variable or character array element that is always assigned the value YES.
UNFORMATTED=unf	unf is a character variable or character array element that is always assigned the value YES.
RECL=rcl	rcl is a integer variable or integer array element that is assigned the value of the record length of the file connected for direct access. If the file is connected for formatted input/output, the length is the number of characters. If the file is connected for unformatted input/ output, the length is in words. If there is no connection or if the connection is not for direct access, rcl becomes undefined.
NEXTREC=nr	nr is a integer variable or array element that is assigned the value n+1, where n is the record number of the last record read or written on the file connected for direct access. If the file is connected but no records have been read or written since the connection, nr is assigned the value 1. If the file is not connected for direct access or if the position of the file is indeterminate because of a previous error condition, nr becomes undefined.
BLANK=blnk	blnk is a character variable or array element that is assigned a value depending on the blank control in effect. The value NULL is assigned if null blank control is in effect for the file connected for formatted input/output, and the value ZERO is assigned if zero blank control is in effect for the file connected for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, blnk becomes undefined.

When an INQUIRE statement for a file is executed the inquiry specifiers NAMED, NAME, SEQUENTIAL, DIRECT, FORMATTED and UNFORMATTED become defined if the file exists. Note that NUMBER becomes defined only if OPENED is true. Also, ACCESS, FORM, RECL, NEXTREC, and BLANK become defined only if OPENED is true.

When an INQUIRE statement for a unit is executed the inquiry specifiers NUMBER, NAMED, NAME, ACCESS, SEQUENTIAL, DIRECT, FORM, FORMATTED, UNFORMATTED, RECL, NEXTREC, and BLANK are assigned a value only if EXIST is true and OPENED is true. EXIST and OPENED always become defined unless an error condition occurs.

Following are examples of the INQUIRE statement. In the examples NXR1 and CHR2 are character variables, NO and NXRC are integer variables and L1 and L2 are logical variables.

```
INQUIRE (FILE="A/B.", EXIST=L1, OPENED=L2, NUMBER=NO)
INQUIRE (UNIT=9, NAMED=L1, NAME=CHR1, NEXTREC=NXRC)
INQUIRE (6, BLANK=CHR1, FORM=CHR2)
```

BACKSPACE STATEMENT

The executable BACKSPACE statement is an auxiliary I/O statement which allows a file to be repositioned.

The two allowable formats for the BACKSPACE statement are:

```
BACKSPACE u
BACKSPACE (alist)
```

where u is an external unit identifier, and
alist is a list of specifiers, separated by commas, that contains exactly one external UNIT specifier and may contain at most one of each of the other specifiers listed below.

[UNIT =] u — is an external unit specifier.

IOSTAT = i — is an I/O status specifier.

ERR = s — is an error specifier.

Execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned before the preceding record. If there is no preceding record, the position of the file is not changed. Note that if the preceding record is an endfile record, the file becomes positioned before the endfile record.

Execution of a BACKSPACE statement containing an I/O status specifier causes i to become defined with a zero value if no error condtion exists or with a positive value corresponding to an error condition if an error condition exists.

The external unit specified by a BACKSPACE statement must be connected for sequential access.

Backspacing over records written using list-directed formatting is prohibited.

The BACKSPACE statement may not reference line printer, card punch or remote files.

Following are examples of the BACKSPACE statement.

```
BACKSPACE 10
BACKSPACE(UNIT = 8, IOSTAT=IORSLT, ERR = 89)
BACKSPACE(9, ERR = 99)
```

ENDFILE STATEMENT

The executable ENDFILE statement is an auxiliary I/O statement which writes an endfile record.

The two allowable formats for the ENDFILE statement are:

```
ENDFILE u
ENDFILE (alist)
```

where u is an external unit identifier, and
alist is a list of specifiers, separated by commas, that contains exactly one external UNIT specifier and may contain at most one of each of the other specifiers listed below.

[UNIT =] u	is an external unit specifier.
IOSTAT = i	is an I/O status specifier.
ERR = s	is an error specifier.

Execution of an ENDFILE statement writes an endfile record as the next record of the file. The file is then positioned after the endfile record. If the file has been connected for direct access, only those records before the endfile record are considered to have been written. Thus, only those records may be read during subsequent direct access connection to the file.

After execution of an ENDFILE statement, a BACKSPACE or REWIND statement must be used to reposition the file prior to execution of any data transfer input/output statement.

Execution of an ENDFILE statement for a file that is connected but does not exist creates the file.

Execution of a ENDFILE statement containing an I/O status specifier causes i to become defined with a zero value if no error condtion exists or with a positive value corresponding to an error condition if an error condition exists.

Following are examples of the ENDFILE statement.

```
ENDFILE (UNIT = 3)
ENDFILE 6
ENDFILE(4,ERR=49)
```

REWIND STATEMENT

The executable REWIND statement is an auxiliary I/O statement which allows a file to be repositioned.

The two allowable formats for the REWIND statement are:

```
REWIND u
REWIND (alist)
```

where u is an external unit identifier, and
alist is a list of specifiers, separated by commas, that contains exactly one external UNIT specifier and may contain at most one of each of the other specifiers listed below.

[UNIT =] u	is an external unit specifier. The definition is specified in the control information list definition.
IOSTAT = i	is an I/O status specifier. The definition is specified in the control information list definition.
ERR = s	is an error specifier. The definition is specified in the control information list definition.

Execution of a REWIND statement causes the specified file to be positioned at its initial point. Note that if the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

Execution of a REWIND statement for a file that is connected but does not exist is permitted but has no effect.

Execution of a REWIND statement containing an I/O status specifier causes i to become defined with a zero value if no error condtion exists or with a positive value corresponding to an error condition if an error condition exists.

Following are examples of the REWIND statement.

```
REWIND (UNIT = 9)
REWIND 9
REWIND (3, ERR=39, IOSTAT=IORSLT)
```

FIND STATEMENT

The FIND statement is used to position a direct access file at a specified record.

The format of the FIND statement is:

```
FIND (findlist)
```

where findlist is of the forms:

```
[UNIT =] u , REC = r
REC = r, UNIT = u
u' r
```

where u is an external unit identifier, and
r is an integer expression whose value is positive. It specifies a record number.

Execution of a FIND statement will position the designated file before the specified record.

The following examples all position unit 5 to record number 7.

```
FIND (5, REC=7)
FIND (UNIT=5,REC=7)
FIND (REC=7,UNIT=5)
FIND (5'7)
```

FILE STATEMENT

A file statement declares a unit and assigns values to certain specifiers and to file attributes for the unit.

The format of a file declaration is:

```
FILE u (filelist)
```

where the characters FILE appear in columns 1-4 followed by at least two blanks,
u is an external unit identifier, and
filelist is a list of specifiers.

At most one of each of the following specifiers may appear in filelist: FILE, STATUS, ACCESS, FORM, RECL, and BLANK. A full description of the specifiers can be found with the OPEN statement. In the FILE statement the expressions used for these specifiers are restricted to only constant expressions. Character constant expressions, integer constant expressions, or whatever is the appropriate constant expression for the specifier.

fa = v	is a file attribute specifier. The definition of this specifier is with the OPEN statement definition.
FILE = n	is a file name specifier. The definition of this specifier is with the OPEN statement definition.
STATUS = sta	is a STATUS specifier. The definition of this specifier is with the OPEN statement definition.
ACCESS =acc	is an ACCESS specifier. The definition of this specifier is with the OPEN statement definition.

FORM = fm is a FORM specifier. The definition of this specifier is with the OPEN statement definition.

RECL =l is a RECL specifier. The definition of this specifier is with the control information list definition.

BLANK = bl is a BLANK specifier. The definition of this specifier is with the OPEN statement definition.

File statements must precede all other statements in a FORTRAN 77 and apply to the entire program.

Following are some examples of the FILE statement.

```
      FILE  3(FILE="WRS/1.",STATUS="OLD",KIND="DISK",RECL=30,
     1  ACCESS="DIRECT")
      FILE  4(KIND="REMOTE", MYUSE="IO")
      FILE  6(KIND="PETAPE",MAXRECSIZE=45,BLOCKSIZE=45,BLANK="ZERO")
```

SECTION 9

FORMAT SPECIFICATIONS

INTRODUCTION

Format specifications are used in conjunction with formatted input/output statements to provide information that directs the editing between the internal representation of data and the character strings of a record or a sequence of records in a file.

A format may be specified explicitly or it may be specified implicitly as a list directed format (by supplying an asterisk). A format is specified explicitly in a FORMAT statement or as the value stored in character arrays, character variables, or other character expressions.

FORMAT STATEMENT

The form of a FORMAT statement is:

```
FORMAT fs
```

where fs is a format specification. This statement must be labeled.

The form of a format specification is:

```
([edlist])
```

where edlist is a list.

The forms of the items in the list are:

```
[r]ed
ned
[r]fs
```

where ed is a repeatable edit descriptor,
ned is a non repeatable edit descriptor,
fs is a format specification with a non-empty edlist, and
r is a nonzero unsigned, integer constant called a repeat specification.

The comma used to separate the items of the edlist may be omitted before or after a slash, before or after a colon edit descriptor, or between a P edit descriptor and an immediately following F, E, D, or G edit descriptor.

Format control is initiated at the beginning of a formatted I/O statement. Each action of format control depends jointly on the next edit descriptor of the current format specification and the next item in the I/O list, if one exists.

If an input/output list specifies at least one list item, at least one repeatable edit descriptor must exist in the format specification. Note that an empty format specification of the form ( ) may be used only if no list items are specified. In this case, one input record is skipped or one output record containing no characters is written. Except for an edit descriptor preceded by a repeat specification, r ed, and a format specification preceded by a repeat specification, r(flist), a format specification is interpreted from left to right. A format specification or edit descriptor preceded by a repeat specification r is processed as a list of r format specifications or edit descriptors. Note that an omitted repeat specification is treated the same as a repeat specification whose value is 1.

For each repeatable edit descriptor interpreted in a format specification, there corresponds one item specified by the I/O list. (A list item of type complex requires the interpretation of two F, E, D, or G edit descriptors.) For each P, X, T, TL, S, SP, SS, H, BN, BZ, slash, colon, quote, or apostrophe edit descriptor, there is no corresponding item specified by the input/output list, and format control communicates information directly with the record.

Whenever format control encounters a repeatable edit descriptor in a format specification, it determines whether there is a corresponding item specified by the input/output list. If there is such an item, it transmits appropriately edited information between the item and the record, and then format control proceeds. If there is no corresponding item, format control terminates. If format control encounters the right-most parenthesis of a complete format specification and another list item is not specified, format control terminates. However, if another list item is specified, the file is positioned at the beginning of the next record and format control then reverts to the beginning of the format specification terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, format control reverts to the first left parenthesis of the format specification. If such reversion occurs, the reused portion of the format specification must contain at least one repeatable edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat specification, the repeat specification is reused. Reversion of format control, in itself, has no effect on the scale factor, the S, SP, or SS edit descriptor sign control, or the BN or BZ edit descriptor blank control.

CARRIAGE CONTROL

Any record which is to be output to a line printer or remote file must contain a prescribed carriage control character in its first character position. This character will not be output but is interpreted as follows:

blank	Single space
0	Double space
1	Page eject
+	No line feed
Digit 2-9	Advance to channel n

Any other character in the first character position will not be output and single spacing will be assumed.

All carriage spacing indicated by the carriage control character is performed prior to output of the line.

EDIT DESCRIPTORS

Edit descriptors are used to specify the form of a record and to direct the editing between the characters in a record and the external representation of data.

An edit descriptor is either a repeatable descriptor or a non-repeatable edit descriptor.

The repeatable edit descriptors are:

I Integer Editing
F Real/Double Precision Editing
E Real/Double Precision Editing
D Real/Double Precision Editing
G Real/Double Precision Editing
L Logical Editing
A Character Editing
Z Hexadecimal Editing

The non-repeatable edit descriptors are:

Quote
Apostrophe
Hollerith
T, TL, TR Positional Editing
X Positional Editing
/ Current Record Termination
Colon Format Termination
S, SP, SS Optional Plus Characters
P Scale Factor
BN, BZ Blank Editing

In the following description a field is a part of a record that is read on input or written on output when format control processes an I, E, F, D, G, L, A, H, apostrophe, or quote edit descriptor. The field width is the size in characters of the field.

APOSTROPHE AND QUOTE EDITING

The apostrophe and quote edit descriptors function exactly alike. They have the form of a character constant.

On output, the characters enclosed in the delimiters, apostrophe or quote, including blanks are written to the output media. The number of characters between the delimiters should be less than or equal to 255. Within the field to be written two consecutive delimiters with no intervening blanks are counted as a single delimiter.

Following are examples of apostrophe and quote editing on output:

Specification	External string
'ABCD'	ABCD
'ISN''T'	ISN'T
"GHIJK"	GHIJK
"QUOTE CHARACTER """	QUOTE CHARACTER "

H EDITING

The form of the H edit descriptor is:

    nHh1h2h3...hn

where: h is a valid EBCDIC character, and
n is an unsigned, nonzero integer.

On output, the H edit descriptor causes character information to be written from the n characters (including blanks) following the H of the edit descriptor.

On input, the n characters following the H of the edit descriptor are replaced by n characters from the input field. The edit descriptor is modified during execution; subsequent use of the edit descriptor for output causes the new characters to be written.

Note that if an H edit descriptor occurs within a character constant and includes an apostrophe, the apostrophe must be represented by two consecutive apostrophes, which are counted as one character in specifying the number of characters of the H edit descriptor.

Following are examples of H editing on output:

Specification	External string
5HX ABC	X ABC
1H5	5
3H'B'	'B'

POSITIONAL EDITING

The T, TL, TR, and X edit descriptors specify the position at which the next character will be transferred to or from the record. These descriptors do not cause characters to be transmitted. If characters are transmitted to positions at or after the position specified by these edit descriptors positions skipped remain unchanged. On output characters already filled may be replaced using positional editing.

T, TL, And TR Editing

The form for the T, TL, TR edit descriptors is:

```
Tc
TLc
TRc
```

where c is an unsigned, nonzero integer constant.

The Tc edit descriptor indicates that the transmission of the next character to or from a record is to occur at the cth character position. Note that for printed output the first print position is used for carriage control, therefore character position c is actual print position c-1.

The TLc edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position c characters backward from the current position. However, if the current position is less than or equal to position c, the TLc edit descriptor indicates that the transmission of the next character to or from the record is to occur at position 1 of the current record.

The TRc edit descriptor indicates that the transmission of the next character to or from the record is to occur at the character position c characters forward from the current position.

X Editing

The form for the X edit descriptor is:

    [n]X

where n is an unsigned, nonzero integer constant.

If n is omitted 1 is implied. The X edit descriptor indicates that the transmission of the next character to or from a record is to occur at the position n characters forward from the current position.

SLASH EDITING

The slash edit descriptor indicates the end of data transfer on the current record.

On input from a file connected for sequential access, the remaining portion of the current record is skipped and the file is positioned at the beginning of the next record. This record becomes the current record. On output to a file connected for sequential access, a new record is created and becomes the last and current record of the file. Note that a record that contains no characters may be written on output.

If the file is an internal file or a file connected for direct access, the record is filled with blank characters. Note also that an entire record may be skipped on input.

For a file connected for direct access, the record number is increased by one and the file is positioned at the beginning of the record that has that record number. This record becomes the current record.

COLON EDITING

The colon edit descriptor terminates format control if there are no more items in the input/output list. It has no effect if there are more items in the list.

S, SP, AND SS EDITING

The S, SP, and SS edit descriptors may be used to control optional plus characters in numeric output fields. At the beginning of execution of a formatted output statement, standard numeric output fields do not include a plus sign. If the SP edit descriptor is encountered in a format specification, the optional plus is generated in any position that normally contains an optional plus. If an SS edit descriptor is encountered then the optional plus is not generated in each subsequent position that normally contains an optional plus. If an S edit descriptor is encountered, the standard option of not producing a plus is again established.

These descriptors only affect I, E, F, D, and G editing on output. They have no effect during the execution of an input statement.

P EDITING

The P edit descriptor specifies a scale factor to be used for scaling numbers input and output using F, E, D, and G edit descriptors.

The format of the P edit descriptor is:

kP

where k is an optionally signed integer constant called the scale factor.

The value of the scale factor is zero at the beginning of execution of each I/O statement. Once encountered a scale factor applies to all subsequently interpreted F, E, D, and G edit descriptors until another scale factor is encountered.

The scale factor affects editing in the following manner:

1. On input, with F, E, D, and G editing, the scale factor has no effect if there is an exponent part in the input field.

2. On input, with E, D, G and F editing, the effect of the scale factor is as follows:

   external quantity=internal value*(10**k)

3. On output, with E and D editing, the scale factor does not effect the external value. It affects the positioning of the decimal point in the output field. The real constant part of the output value is multiplied by 10**k and the exponent is reduced by k.

4. On output, with G editing, the scale factor is suspended if F editing is to be used. If E editing is to be performed, the scale factor has the same effect as with E output editing.

5. On output F editing the effect of the scale factor is as follows:

   external quantity=internal value*(10**k)

Following are examples of the scale factor usage:

	Edit Descriptor	Internal Value	Input/Output Field column 1234567890
INPUT	2PF5.1	12.345	12345
	2PE5.1	12000.	1.2E4
	2PG10.2	.1234	1234
	-1PF6.2	123.4	1234
OUTPUT	2PF10.2	1.234	123.40
	-2PF10.2	1.234	0.01
	1PE10.2	1.234	1.23E+00
	-1PD10.2	1.234	.01D+02

BN AND BZ EDITING

The BN and BZ edit descriptors may be used to specify the interpretation of blanks, other than leading blanks, in numeric input fields. At the beginning of execution of each formatted input statement, such blank characters are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier currently in effect for the unit. If a BN edit descriptor is encountered in a format specification, all such blank characters in succeeding numeric input fields are ignored. The effect of ignoring blanks is to treat the input field as if blanks had been removed, the remaining portion of the field right-justified, and the blanks replaced as leading blanks. However, a field of all blanks has the value zero. If a BZ edit descriptor is encountered in a format specification, all such blank characters in succeeding numeric input fields are treated as zeros.

The BN and BZ edit descriptors affect only I, F, E, D, and G editing during execution of an input statement. They have no effect during execution of an output statement.

NUMERIC EDITING

The I, F, E, D, and G edit descriptors are used to specify input/output of integer, real, double precision, and complex data. The following general rules apply:

1. On input, leading blanks are not significant. The interpretation of blanks, other than leading blanks, is determined by a combination of any BLANK= specifier and any BN or BZ blank control that is currently in effect for the unit. Plus signs may be omitted. A field of all blanks is considered to be zero.

2. On input, with F, E, D, and G editing, a decimal point appearing in the input field overrides the portion of an edit descriptor that specifies the decimal point location.

3. On output, positive or zero internal values may be prefixed with a plus, as controlled by the S, SP, and SS edit descriptors. The representation of a negative internal value in the field is prefixed with a minus.

4. On output, the representation of numeric data is right-justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks will be inserted in the field.

5. On output, if the number of characters produced exceeds the field width or if an exponent exceeds its specified length using the Ew. dEe or Gw.dEe edit descriptor, asterisks are placed in the entire field width.

Complex datum, because it consists of a pair of real values, requires two successive F, E, D, or G edit descriptors. The first of the edit descriptors specifies the real part; the second specifies the imaginary part. The two edit descriptors may be different. Note that non-repeatable edit descriptors may appear between the two successive F, E, D, or G edit descriptors.

Integer Editing

The I edit descriptor is used for editing integer data. The I/O list item must be of type integer.

The formats for the I edit descriptors are:

Iw
Iw.m

where w is a nonzero, unsigned, integer constant denoting field width,
m is an unsigned integer constant denoting the minimum number of digits required on output.

On input, an Iw.m edit descriptor is treated identically to an Iw edit descriptor. In the input field, the character string must be in the form of an optionally signed integer constant, except for the interpretation of blanks.

The output field for the Iw edit descriptor consists of zero or more leading blanks followed by a minus if the value of the internal datum is negative, or an optional plus, followed by the magnitude of the internal value in the form of an unsigned integer constant without leading zeros. Note that an integer constant always consists of at least one digit.

The output field for the Iw.m edit descriptor is the same as for the Iw edit descriptor, except that the unsigned integer constant consists of at least m digits and, if necessary, has leading zeros. The value of m must not exceed the value of w. If m is zero and the value of the internal datum is zero, the output field consists of only blank characters, regardless of the sign control in effect.

Following are examples of the I edit descriptors:

	Edit Descriptor	Internal Value	Output/Input Field
		columns	123456789
OUTPUT	I6	1259839	******
	I3	2	2
	I9.3	11	011
	I5	-23	-23
INPUT	I4	123	123
	I2	- 1	-1
	I5	21000	21000

F Editing

The F edit descriptor is specified for editing real, double precision, and complex data.

The form for the F edit descriptor is:

    Fw.d

where w is a nonzero, unsigned, integer constant denoting field width,
d is the number of digits in the fractional part.

On input, the input field should consist of an optional sign, followed by a string of digits optionally containing a decimal point. If the decimal point is omitted, the right-most d digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented. The basic form may be followed by an exponent of one of the following forms:

1. Signed integer constant.

2. E followed by zero or more blanks, followed by an optionally signed integer constant.

3. D followed by zero or more blanks, followed by an optionally signed integer constant.

An exponent containing a D is processed identically to an exponent containing an E.

On output, the field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus, followed by a string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the established scale factor and rounded to d fractional digits. Leading zeros are not permitted except for an optional zero immediately to the left of the decimal point if the magnitude of the value in the output field is less than one. The optional zero must appear if there would otherwise be no digits in the output field.

Following are examples:

	Edit Descriptor	Internal Value		Output/Input Field
			columns	12345678901
OUTPUT	F7.3	36.7929		36.793
	F9.3	36.7934		36.793
	F6.3	- .0316		-.032
	F4.2	579.645		****
	F6.2	579.645		579.65
INPUT	F8.4	3672.5931		36725931
	F10.2	3.672593		3.672593
	F11.1	-367259.		-367259.
	F9.3	-367200.		-3672.E2
	F8.4	.3672		3.672E-1
	F7.2	-34.56		-3456

E And D Editing

The E and D edit descriptors are specified for editing real, double precision, and complex type datum.

The form for D and E edit descriptors is:

```
Ew.d
Ew.dEe
Dw.d
```

where w is a nonzero, unsigned, integer constant denoting field width,
d is an unsigned integer constant denoting the fractional part, and
e is a nonzero, unsigned, integer constant denoting the number of digits in the exponent part. The e has no effect on input.

On input, the field should consist of an optional sign followed by a string of digits optionally containing a decimal point and exponent. If the decimal point is omitted, a decimal point is implied d places to the left of the E or D denoting the exponent, or if there is no exponent, d places from the right edge of the input field.

The form of the output field for a scale factor of zero is:

```
[+ or -] [0] . x1x2...xd exp
```

where + signifies a plus and - a minus sign,
x1x2...xd are the d most significant digits after rounding, and
exp is a decimal exponent of one of the following forms:

Edit Descriptor	Absolute Value of Exponent	Form of Exponent
Ew.d	exp < = 99	Esz1z2
	99 < exp < = 999	sz1z2z3
Ew.dEe	exp < = (10**e)-1	Esz1z2...ze
Dw.d	exp < = 99	Dsz1z2
	99< exp < = 999	sz1z2z3

where z is a digit and s is the required sign of the exponent. A plus sign must be used if the exponent value is zero. The forms Ew.d and Dw.d must not be used if exp > 999.

The scale factor k controls the decimal normalization. If d < k < 0, the output field contains exactly k leading zeros and d - k significant digits after the decimal point. If 0 < k < d + 2, the output field contains exactly k significant digits to the left of the decimal point and d - k + 1 significant digits to the right of the decimal point. Other values of k are not permitted.

Examples of E and D edit descriptors follow:

	Edit Descriptor	Internal Value	Input/Output Field column 123456789012345678
INPUT	E11.4	2.5046	25046
	E11.4	- .0025046	-25046E-3
	E9.3	2.77	2.77
	E8.5	1268000	1268E+3
	D18.3	413679.7	413679.7
	D8.1	-305.E35	-305D36
OUTPUT	E13.5	36.7929	.36793E+02
	E10.5	-36.7929	**********
	E11.5	-36.7929	-.36793E+02

G EDITING

The G edit descriptor may be used with list items of type integer, logical, real, complex and double precision.

The format for the G edit descriptor is:

Gw.d
Gw.dEe

where w is a nonzero, unsigned, integer constant denoting field width,
d is an unsigned integer constant denoting the fractional part, and
e is a nonzero, unsigned, integer constant denoting the number of digits in the exponent part. The e has no effect on input.

When the Gw.d edit descriptor is used with a list item of type integer, the datum is edited with an Iw edit descriptor.

When the Gw.d edit descriptor is used with a list item of type logical, the datum is edited with an Lw edit descriptor.

G input editing of real and double precision list items is the same as for F editing of these list items.

For output the method of representation depends on the magnitude of the datum being edited. For v<.1 or v>10**d, E editing is used. Otherwise if v is the absolute value of the list item, then for k = 0, 1, 2,...d, if 10 **(d-k-1)<v<10**(d-k), then the format F(w-n).k,nX is used. In the preceding format n is 4 for Gw.d editing and n is e+2 for Gw.dEe editing.

A scale factor affects the G format only if E editing is used.

Following are examples of the G format:

	List Type	Edit Descriptor	Internal Value	Input/Output Field column 1234567890
INPUT	logical	G5.2	.TRUE.	.T
	real	G10.3	5.679	5.679
	integer	G10.5	325672	325672
	real	G10.3	360	3.6E2
OUTPUT	real	G10.2	95	95.
	real	G10.2	.9	0.90
	real	G10.2	300	0.30E+03
	real	G10.2E3	.35	0.35
	integer	G10.9	1362	1362

L EDITING

The L edit descriptor is specified to edit logical type datum.

The form for the L edit descriptor is:

Lw

where w is a nonzero, unsigned, integer constant denoting field width.

On input, the list item becomes defined with a logical value. The field should consist of optional blanks, optionally followed by a decimal point, followed by a T for true or F for false. The T or F may be followed by additional characters in the field. Note that the logical constants .TRUE. and .FALSE. are acceptable input forms.

On output, the field consists of w - 1 blanks followed by a T or F, as the value of the internal datum is true or false, respectively.

Examples of L editing follow:

	Edit Descriptor	Internal Value	Output/Input Field column 123
INPUT	L1	.TRUE.	T
	L3	.FALSE.	.F
OUTPUT	L1	.FALSE.	F
	L3	.FALSE.	F
	L2	.TRUE.	T

A EDITING

The A edit descriptor is used with input/output list items of type character. On input, the list item becomes defined with character data. On output, the list item must be defined with character data.

The form for A editing is:

A [w]

where w is a nonzero, unsigned, integer constant denoting field width.

If the field width is not specified with the edit descriptor, the number of characters in the field is equal to the length of the character data.

On input, if the field width specified exceeds the length of the character data, the right-most characters equal to the length of the list item are stored. If the field width is less than the length of the character data item, the characters of the input field are stored left-justified with trailing blanks into the list item.

On output, if the field width is greater than the length of the character data item, the data item is placed right-justified in the field preceded by blanks. If the field width is less than or equal to the length of the list item, the w left-most characters are placed in the field.

Following are examples:

	Character Item Length	Edit Descriptor	Internal Value	Input/Output Field column 123456
INPUT	5	A3	ABC	ABC
	3	A3	ABC	ABC
	2	A3	BC	ABC
OUTPUT	6	A3	ABCDEF	ABC
	2	A6	AB	AB
	5	A5	ABCDE	ABCDE

Z EDITING

The Z edit descriptor is used with I/O list items of a type other than character. On input, the list item becomes defined with a hexadecimal value. On output, the list item is outputted as a string of hexadecimal digits.

The form for the Z edit descriptor is:

Zw

where w is a nonzero, unsigned, integer constant denoting field width.

The input field is interpreted as w hexadecimal digits. Leading, trailing, and embedded blanks are interpreted as zeros, and no sign must appear in the field. If the list element is single precision and the input datum is less than 12 hexadecimal digits long, it is stored right-justified with zero fill in the storage word. If the list element is double precision, and the input datum is 12 or fewer digits, it is stored right-justified with zero fill in the first word, and zeros are stored in the second word. If the input datum is more than 12 digits, the first 12 digits are stored into the first word and the remaining digits are stored right-justified with zero fill in the second word. If the field width on input is greater than the 12 or 24 digits that can be stored in a single or double precision word respectively, the right-most 12 or 24 digits are taken from the field.

On output, if the field width specified is greater than 12 (24 for double precision) the 12 (or 24) digits are output right-justified with blank fill within the field. If the field width is less than the internal representation the right-most hexadecimal digits are transferred to fill the field width.

Examples follow:

	List Item Type	Edit Descriptor	Internal Value	Input/Output Field column 123456789012345
INPUT	S	Z2	Z00000000006F	6F
	S	Z15	Z4567890ABCDE	1234567890ABCDE
	D	Z15	Z1234567890AB 000000000CDE	1234567890ABCDE
OUTPUT	S	Z5	Z0000567890AB	890AB
	S	Z12	Z1234567890AB	1234567890AB
	S	Z15	Z1234567890AB	1234567890AB

FORMAT SPECIFICATIONS IN CHARACTER ENTITY

If the format identifier in a formatted input/output statement is a character array name, character variable name, or other character expression, the left-most character positions of the specified entity must be in a defined state with character data that constitute a format specification when the statement is executed.

A character format specification must be of the form described in the preceding paragraphs. Note that the form begins with a left parenthesis and ends with a right parenthesis. Character data may follow the right parenthesis that ends the format specification and blank characters may precede the format specification with no effect on the format specification.

If the format identifier is a character array name, the length of the format specification may exceed the length of the first element of the array; a character array format specification is considered to be a concatenation of all the array elements of the array in the order given by array element ordering. However, if a character array element name is specified as a format identifier, the length of the format specification must not exceed the length of the array element.

Following are examples of format specified as values of character arrays, character variables, or other character expressions:

```
CHARACTER FMT1*20
DATA FMT1/"(I2,F10.5,5X,I3)"/
WRITE (6,FMT=FMT1)I,X,J
WRITE (6,FMT="(F5.1)")Z
WRITE (6,FMT="("//FMT1(2:9)//")") I,X
```

LIST-DIRECTED FORMAT

The use of list directed format (or free-format) with an input/output statement allows I/O to be performed with editing, but without requiring the user to explicitly specify a format specification. List-directed formatting is invoked with an asterisk, which is a format specifier.

The characters in one or more list-directed records constitute a sequence of values and value separators. The end of a record has the same effect as a blank character, unless it is within a character constant. Any sequence of two or more consecutive blanks is treated as a single blank, unless it is within a character constant.

Each value is either a constant, a null value, or of one of the following forms:

```
r*c
r*
```

where r is an unsigned, nonzero, integer constant.

The r*c form is equivalent to r successive appearances of the constant c, and the r* form is equivalent to r successive null values. Neither of these forms may contain embedded blanks, except where permitted within the constant c.

A value separator is one of the following:

1. A comma optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks.

2. A slash optionally preceded by one or more contiguous blanks and optionally followed by one or more contiguous blanks.

3. One or more contiguous blanks between two constants or following the last constant.

Input forms acceptable to format specifications for a given type are acceptable for list-directed formatting, except as noted. The form of the input value must be acceptable for the type of the input list item. Blanks are never used as zeros, and embedded blanks are not permitted in constants, except within character constants and complex constants as specified below. Note that the end of a record has the effect of a blank, except when it appears within a character constant.

When the corresponding input list item is of type real or double precision, the input form is that of a numeric input field. A numeric input field is a field suitable for F editing which is assumed to have no fractional digits unless a decimal point appears within the field.

When the corresponding list item is of type complex, the input form consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma, and followed by a right parenthesis. The first numeric input field is the real part of the complex constant and the second is the imaginary part. Each of the numeric input fields may be preceded or followed by blanks. The end of a record may occur between the real part and the comma or between the comma and the imaginary part.

When the corresponding list item is of type logical, the input form must not include either slashes or commas among the optional characters permitted for L editing.

When the corresponding list item is of type character, the input form consists of a valid character constant. The character constants may be continued from the end of one record to the beginning of the next record and may be continued on as many records as needed. If the length of the list item and the length of the constant are not equal, the list item is assigned the value of the constant as though the constant were assigned the list item in a character assignment statement.

A null value may be specified by having no characters between successive value separators, no characters preceding the first value separator in the first record read by each execution of a list-directed input statement, or the r* form. A null value has no effect on the definition status of the corresponding input list item. If the input list item is defined, it retains its previous value; if it is undefined, it remains undefined. A null value may not be used as either the real or imaginary part of a complex constant, but a single null value may represent an entire complex constant. Note that the end of a record following any other separator, with or without separating blanks, does not specify a null value.

A slash encountered in the data value list during execution of a list-directed input statement causes termination of execution of that input statement after assignment of the previous value. If there are additional items in the input list, they are assigned null values.

On output, each value is edited into the appropriate format. With the exception of character constants, the values are separated by a comma value separator.

New records begin as necessary, but, except for complex constants and character constants, the end of a record does not occur within a constant, and blanks do not occur within a constant.

Logical output constants are T for the value true and F for the value false.

Integer output constants are produced with the effect of an I edit descriptor.

Real and double precision constants are produced with the effect of either an F edit descriptor or an E edit descriptor, depending on the magnitude of the value.

Complex constants are enclosed in parentheses, with a comma separating the real and imaginary parts. The end of a record may occur between the comma and the imaginary part only if the entire constant is as long as, or longer than, an entire record.

Character constants produced are not delimited by apostrophes nor quotes and are not preceded or followed by a comma.

Each output record begins with a blank character to provide carriage control when the record is printed.

Following is an example of list-directed formatting on input:

```
LOGICAL LG(5)
CHARACTER CHAR1*4, CHAR2*2, CHAR3*3
DOUBLE PRECISION DBLE
COMPLEX CMPLX
READ(5,FMT=*)X,Y,DBLE,LG,CMPLX,CHAR1,INT1,INT2,CHAR2,CHAR3,Z
```

Input data for the above READ statement follows:

```
  5.9, 6.3E10, 367.D23, 2*.TRUE.,  , 2*.F, (1.1,1.1)
"AB""C", 22, 22, "ABCD", "AB" /
```

The following assignments take place as a result of the preceding READ statement with the indicated data:

```
X = 5.9
Y = 6.3E10
DBLE = 367.D23
LG(1) = .TRUE.
LG(2) = .TRUE.
LG(4) = .FALSE.
LG(5) = .FALSE.
CMPLX = (1.1,1.1)
CHAR1 = "AB""C     A value of AB"C
CHAR2 = "AB"
CHAR3 = "AB "
INT1 = 22
INT2 = 22
```

The value of Z and LG(3) do not change.

Following is an example of list-directed formatting on output:

```
WRITE (6,*)X, INT1, LG(1), LG(5), CHAR1, CHAR1
```

Following is the output produced by the above WRITE statement assuming the values of the variables were assigned by the preceding list-directed READ statement.

```
5.9,22,T,F,2*AB"C
```

SECTION 10

MAIN PROGRAM

INTRODUCTION

Every FORTRAN77 program must have one main program. Execution of a program begins with the first executable statement in this program unit.

A main program is a program unit which does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement. It may have a PROGRAM statement as its first statement. Main programs may not contain RETURN or ENTRY statements and the appearance of a SAVE statement in this program unit has no effect.

PROGRAM STATEMENT

This statement is used to assign a symbolic name to a main program. The format of a PROGRAM statement is:

```
PROGRAM n
```

where n is a symbolic name.

This is an optional statement and when used must be the first statement of the program unit.

The symbolic name assigned to the main program is global to the executable program. Therefore care must be taken that the name selected is not the same as that of an external procedure, block data subprogram, common block, or local name in the main program.

SECTION 11

FUNCTIONS AND SUBROUTINES

INTRODUCTION

Functions and subroutines, called procedures, consist of four types: subroutines, external functions, statement functions, and intrinsic functions.

Intrinsic functions, statement functions, and external functions are referred to collectively as functions.

External functions and subroutines are referred to collectively as external procedures. These external procedures are defined in program units called subprograms: function subprograms and subroutine subprograms.

FUNCTIONS

Functions are procedures which are referenced in expressions and supply a value to the expression. The value supplied is the value of the function.

There are three types of functions:

1. Intrinsic functions.
2. Statement functions.
3. External functions.

A function reference is used to reference an intrinsic function, statement function, or external function.

The format of a function reference is:

```
f ( [a [,a]...] )
```

where f is the name of a function or a dummy procedure, and
a is an actual argument.

The type of the result of a statement function or external function reference is the same as the type of the function name. The type is specified in the same manner as for variables and arrays. The type of the result of an intrinsic function is specified in table 11-1.

INTRINSIC FUNCTIONS

Intrinsic functions are function subprograms which are known to the compiler and need not be supplied in the program. These functions are identified by two types of names: specific names and generic names. An IMPLICIT statement does not change the type of an intrinsic function. The specific names that identify the intrinsic functions, their generic names, function definitions, and the type of results are illustrated in table 11-1. The order, number, and type of the actual arguments passed to the intrinsics must agree with the specifications in the table when the specific name is used to reference a function.

For those intrinsic functions that have more than one argument, all arguments must be of the same type.

Generic names simplify the use of intrinsic functions. The generic function name may be used with various types of arguments. If the generic name is used, the type of the result is the same as the type of the argument. The exceptions to this are those functions which do type conversion, nearest integer, and absolute value with a complex argument.

An actual argument in an intrinsic function reference may be any expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant.

A specific name of an intrinsic function that appears in an INTRINSIC statement may be used as an actual argument in an external procedure reference. However, the names of intrinsic functions for type.conversion, lexical relationship, for choosing the largest or smallest value, TIME, and DATE must not be used as actual arguments. Note that such an appearance does not cause the intrinsic function to be classified as an external function.

Table 11-1. Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Type Conversion	Conversion	1	INT	—	Integer	Integer
	to Integer			INT	Real	Integer
	int (N)			IFIX	Real	Integer
	See Note 1.			IDINT	Double	Integer
				—	Complex	Integer
	Conversion	1	REAL	REAL	Integer	Real
	to Real			FLOAT	Integer	Real
	See Note 2.			—	Real	Real
				SNGL	Double	Real
				—	Complex	Real
	Conversion	1	DBLE	—	Integer	Double
	to Double			—	Real	Double
	See Note 3.			—	Double	Double
				—	Complex	Double
	Conversion	1 or 2	CMPLX	—	Integer	Complex
	to Complex			—	Real	Complex
	See Note 4.			—	Double	Complex
				—	Complex	Complex
	Conversion	1		ICHAR	Character	Integer
	to Integer					
	See Note 5.					
	Conversion	1		CHAR	Integer	Character
	to Character					
	See Note 5.					
Truncation	int (N)	1	AINT	AINT	Real	Real
	See Note 1.			DINT	Double	Double
Nearest Whole Number	int(N+ .5) if N ⩾ 0	1	ANINT	ANINT	Real	Real
	int(N- .5) if N < 0			DNINT	Double	Double
Nearest Integer	int(N+ .5) if N ⩾ 0	1	NINT	NINT	Real	Integer
	int(N-.5) if N < 0			IDNINT	Double	Integer
Absolute Value	N	1	ABS	IABS	Integer	Integer
				ABS	Real	Real
	See Note 6.			DABS	Double	Double
	$(N_r 2 + N_r 2)$ '/2			CABS	Complex	Real
Remaindering	N1−int(N1/N2)* N1	2	MOD	MOD	Integer	Integer
	See Note 1			AMOD	Real	Real
	and Note 20.			DMOD	Double	Double

Table 11-1. Intrinsic Functions (Cont.)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Transfer of Sign	N1 if N2 ⩾ 0 – N1 if N2 < 0 See Note 21.	2	SIGN	ISIGN	Integer	Integer
				SIGN	Real	Real
				DSIGN	Double	Double
Positive Difference	N1 – N2 if N1 > N2 0 if N1 ⩽ N2	2	DIM	IDIM	Integer	Integer
				DIM	Real	Real
				DDIM	Double	Double
Double Precision Product	N1*N2	2		DPROD	Real	Double
Choosing Largest Value	max (N1, N2, . . .)	⩾ 2	MAX	MAX0	Integer	Integer
				AMAX1	Real	Real
				DMAX1	Double	Double
				AMAX0	Integer	Real
				MAX1	Real	Integer
Choosing Smallest Value	min (N1, N2, . . .)	⩾ 2	MIN	MIN0	Integer	Integer
				AMIN1	Real	Real
				DMIN1	Double	Double
				AMIN0	Integer	Real
				MIN1	Real	Integer
Length	Length of Character Entity	1		LEN	Character	Integer
Index of a Substring	Location of Substring N2 in String N1 See Note 10.	2		INDEX	Character	Integer
Imaginary Part of Complex Argument	ai See Note 6.	1		AIMAG	Complex	Real
Conjugate of a Complex Argument	(N_r – Ni) See Note 6.	1		CONJG	Complex	Complex
Square Root	(N) 1/2 See Note 22.	1	SORT	SORT	Real	Real
				DSORT	Double	Double
				CSORT	Complex	Complex
Exponential	e**N	1	EXP	EXP	Real	Real
				DEXP	Double	Double
				CEXP	Complex	Complex
Natural Logarithm	LOG(N) See Note 23.	1	LOG	ALOG	Real	Real
				DLOG	Double	Double
				CLOG	Complex	Complex

Table 11-1. Intrinsic Functions (Cont.)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Common Logarithm	log 10(N)	1	LOG10	ALOG10	Real	Real
	See Note 23.			DLOG10	Double	Double
Sine	sin(N)	1	SIN	SIN	Real	Real
				DSIN	Double	Double
	See Note 24.			CSIN	Complex	Complex
Cosine	cos(N)	1	COS	COS	Real	Real
				DCOS	Double	Double
	See Note 24.			CCOS	Complex	Complex
Tangent	tan(N)	1	TAN	TAN	Real	Real
	See Note 24.			DTAN	Double	Double
Arcsine	arcsin(N)	1	ASIN	ASIN	Real	Real
	See Note 25.			DASIN	Double	Double
Arccosine	arccos(N)	1	ACOS	ACOS	Real	Real
	See Note 26.			DACOS	Double	Double
Arctangent	arctan(N)	1	ATAN	ATAN	Real	Real
	See Note 27.			DATAN	Double	Double
	arctan(N1/N2)	2	ATAN2	ATAN2	Real	Real
	See Note 27.			DATAN2	Double	Double
Hyperbolic Sine	sinh(N)	1	SINH	SINH	Real	Real
				DSINH	Double	Double
Hyperbolic Cosine	cosh(N)	1	COSH	COSH	Real	Real
				DCOSH	Double	Double
Hyperbolic Tangent	tanh(N)	1	TANH	TANH	Real	Real
				DTANH	Double	Double
Lexically Greater Than or Equal	N1 > N2 See Note 12.	2		LGE	Character	Logical
Lexically Greater Than	N1 > N2 See Note 12.	2		LGT	Character	Logical
Lexically Less Than or Equal	N1 ≤ N2 See Note 12.	2		LLE	Character	Logical
Lexically Less Than	N1 < N2 See Note 12.	2		LLT	Character	Logical

Table 11-1. Intrinsic Functions (Cont.)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Time	See Note 13.	1		TIME	Character Constant	Real
Date	See Note 14.	1		DATE	Character Constant	Char*6
Pseudo Random Number	See Note 15.	1		RANDOM	Integer	Real
Bit Manipulation	See Note 16.	2	AND	—	Integer Real Logical	Integer Real Logical
	See Note 17.	1	COMPL	—	Integer Real Logical	Integer Real Logical
	See Note 18.	2	EQUIV		Integer Real Logical	Integer Real Logical
	See Note 19.	2	0R		Integer Real Logical	Integer Real Logical

Notes for the table:

1. For n of type integer, int(n)=n. For n of type real or double precision, there are two cases: if n > 1, int(n) is the integer whose magnitude is the largest integer that does not exceed the magnitude of n and whose sign is the same as the sign of n. For example:

   int(-3.7) = -3

   For n of type complex, int(n) is the value obtained by applying the above rule to the real part of n. For n of type real, IFIX(n) is the same as INT(n).

2. For n of type real, REAL(n) is n. For n of type integer or double precision,REAL(n) is as much precision of the significant part of n as n real datum can contain. For n of type complex, REAL(n) is the real part of n. For n of type integer, FLOAT(n) is the same as REAL(n).

3. For n of type double precision, DBLE(n) is n. For n of type integer or real, DBLE(n) is as much precision of the significant part of n as a double precision datum can contain. For n of type complex, DBLE(n) is as much precision of the significant part of the real part of n as a double precision datum can contain.

4. CMPLX may have one or two arguments. If there is one argument, it may be of type integer, real, double precision, or complex. If there are two arguments, they must both be of the same type and may be of type integer, real, or double precision. For n of type complex, CMPLX(n) is n. For n of type integer, real, or double precision, CMPLX(n) is the complex value whose real part is REAL(n) and whose imaginary part is zero. CMPLX(n1,n2) is the complex value whose real part is REAL(n1) and whose imaginary part is REAL(n2).

5. ICHAR provides a means of converting from a character to an integer based on the position of the' character in the collating sequence. The first character in the collating sequence corresponds to position 0 and the last to position n-1, where n is the number of characters in the collating sequence. The value of ICHAR(n) is an integer in the range:0 < ICHAR(N) < n-1, where n is an argument of type character of length 1. The value of n must be in the collating sequence. The position of that character in the collating sequence is the value of ICHAR. For any characters c1 and c2 capable of representation in the processor, (c1 .LE. c2) is true if and only if (ICHAR(c1) .LE. ICHAR(c2)) is true, and (c1 .EQ. c2) is true if and only if (ICHAR(c1) .EQ. ICHAR(c2)) is true. CHAR(i) returns the character in the ith position of the processor collating sequence. The value is of type character of length one. i must be an integer expression whose value must be in the range 0 < i < n-1.

   ICHAR(CHAR(i)) = i for 0 < i < n-1.
   CHAR(ICHAR(c)) = c for any character in the collating sequence.

6 A complex value is expressed as an ordered pair of reals, (ar,ai), where ar is the real part and ai is the imaginary part.

7. All angles are expressed in radians.

8. The result of a function of type complex is the principal value.

9. All arguments in an intrinsic function reference must be of the same type.

10. INDEX(n1,n2) returns an integer value indicating the starting position within the character string n1 of a substring identical to string n2. If n2 occurs more than once in n1, the starting position of the first occurrence is returned. If n2 does not occur in n1, the value zero is returned. Note that zero is returned if LEN(n1) < LEN(n2).

11. The value of the argument of the LEN function need not be defined at the time the function reference is executed.

12. LGE(n1,n2) returns the value true if n1=n2 or if n1 follows n2 in the collating sequence described in American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

    LGT(n1,n2)) returns the value true if n1 follows n2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

    LLE(n1,n2) returns the value true if n1=n2 or if n1 precedes n2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

    LLT(n1,n2) returns the value true if n1 precedes n2 in the collating sequence described in ANSI X3.4-1977 (ASCII), and otherwise returns the value false.

    If the operands for LGE, LGT, LLE, and LLT are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

13. TIME(c) returns time functions in units of seconds where character constant c determines which time function is returned as follows:

Character constant	Description
'DAY'	Current time of day
'ELAPSED'	Elapsed time clock for program
'PROCESSOR'	Total processor time for program
'I/O'	Total Input/Output time for program

14. DATE(c) returns date functions where character constant c determines which date function is returned as follows:

Character constant	Description
'MMDDYY'	Current date in the form MMDDYY
'YYMMDD'	Current date in the form YYMMDD
'YYDDD'	Current date in the form YYDDD
'HHMMSS'	Current time of day in the form HHM

15. The intrinsic function RANDOM returns a uniformly distributed pseudo-random number between 0.0 and 1.0. The actual argument is modified. Each reference to the RANDOM intrinsic in an arithmetic expression must be evaluated whenever the expression itself is evaluated.

16. The intrinsic function AND returns the full word logical product of all data bits of the first argument with corresponding data bits of the second argument. Each argument myst be defined with hexadecimal data.

17. The intrinsic function COMPL returns the full word logical complement of all data bits of its argument. Each argument must be defined with hexadecimal data.

18. The intrinsic function EQUIV returns the full word logical equivalence of all data bits of the first argument with corresponding data bits of the second argument. The bit pattern returned has 1 for each bit position in which the arguments agree, and 0 where they are different. Each argument must be defined with hexadecimal data.

19. The intrinsic function OR returns the full word logical sum of all data bits of the first argument with corresponding data bits of the second argument. Each argument must be defined with hexadecimal data.

20. Remaindering: The result for MOD, AMOD, and DMOD is undefined when the value of the second argument is 0.

21. Transfer of Sign: If the value of the first argument of ISIGN, SIGN, or DSIGN is 0, the result is 0, which is neither positive or negative.

22. Square Root: The value of the argument of SQRT and DSQRT must be greater than or equal to 0. The result of CSQRT is the principal value with the real part greater than or equal to 0. When the real part of the result is 0, the imaginary part is greater than or equal to 0.

23. Logarithms: The value of the argument of ALOG, DLOG, ALOG10, and DLOG10 must be greater than 0. The value of the argument of CLOG must not be (0.,0.). The range of the imaginary part of the result of CLOG is: -PI< imaginary part <= PI. The imaginary part of the result is PI only when the real part of the argument is less than 0 and the imaginary part of the argument is 0.

24. Sine, Cosine, and Tangent: The absolute value of the argument of SIN, DSIN, COS, DCOS, TAN, and DTAN is not restricted to be less than 2PI.

25. Arcsine: The absolute value of the argument of ASIN and DASIN must be less than or equal to 1. The range of the result is: -PI/2 <= result <= PI/2.

26. Arccosine: The absolute value of the argument of ACOS and DACOS must be less than or equal to 1. The range of the result is: 0 <= result <= PI.

27. Arctangent: The range of the result for ATAN and DATAN is: -PI/2 <= result <= PI/2. If the value of the first argument of ATAN2 or DATAN2 is positive, the result is positive. If the value of the first argument is 0, the result is 0 if the second argument is positive, and PI if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is 0, the absolute value of the result is PI/2. The arguments must not both have the value 0. The range of the result for ATAN2 and DATAN2 is: -PI < result <= PI.

STATEMENT FUNCTION

A statement function is a procedure which is defined in a single statement similar to an assignment statement. The statement function may be referenced only within the program unit in which it is defined.

The format of a statement function statement is:

```
f ([d[,d]...])=e
```

where: f is the function name,
d is a dummy argument, (It must be a variable.)
e is an expression.

The relationship between f and e must conform to the assignment rules for arithmetic, logical, and character assignment statements. Note that the type of the expression may be different from the type of the statement function.

The name of a statement function is a local name and must not be the same as the name of any other entity in the program unit except the name of a common block.

The length specification of a character statement function or statement function dummy argument of type character must be an integer constant expression.

A statement function definition must appear after the specification statements and before the first executable statement of a program unit. This is a nonexecutable statement.

The type of value returned when a statement function is referenced depends upon the type associated with the function name. A default type is associated with this name in the same manner as for a variable. This default type may be changed by the appearance of the function name in a TYPE statement or by the implicit typing of the name in an IMPLICIT statement in the same program unit.

The statement function dummy argument list serves only to indicate order, number, and type of arguments for the statement function. The variable names that appear as dummy arguments have a scope of only that statement. A given symbolic name may appear only once in any statement function dummy argument list. An external function reference in the expression which defines the statement function may not redefine dummy arguments.

The expression for defining the statement function may contain constants, symbolic names of constants, variables or array element references, or function references. If a statement function is referenced in the definition, it must appear in preceding lines of the program unit.

Each variable reference may be either a reference to a dummy argument of the statement function or a reference to a variable that appears within the same program unit as the statement function statement.

If a statement function dummy argument name is the same as the name of another entity, the appearance of that name in the expression of a statement function statement is a reference to the statement function dummy argument. A dummy argument that appears in a FUNCTION or SUBROUTINE statement may be referenced in the expression of the statement function statement within the subprogram. A dummy argument that appears in an ENTRY statement that precedes a statement function statement may be referenced in the expression of the statement function within the subprogram.

When a statement function reference is executed, its actual arguments must be defined.

Following are examples of statement functions:

```
LOGICAL COMP
COMP(A) = A.LT.7.5 .OR. A.GT.10.6

ZTOTAL(A,B,C) = A + 2*B + 3*C

YFUNC(X) = X/3.6 + TOT
ZFUNC(X) = YFUNC(X) + X**2
```

EXTERNAL FUNCTIONS

An external function is a procedure that is specified external to the program unit that references it. It is specified in a function subprogram.

A function subprogram is a sequence of statements initiated by a function statement and terminated by an END statement. A function subprogram specifies one or more external functions. The FUNCTION statement indicates the beginning of a function subprogram and specifies the dummy arguments employed in that subprogram to obtain the function value.

The format for a FUNCTION statement is:

```
[t] FUNCTION f ( [d [,d] ...])
```

where: t is one of INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER [*l] where l is the length specification of the result of the character function. l may have any of the forms allowed in a CHARACTER statement except that an integer constant expression must not include the symbolic name of a constant. If a length is not specified in a CHARACTER FUNCTION statement, the character function has a length of one.

f is the symbolic name of the function subprogram in which the FUNCTION statement appears. f is an external function name.

d is a dummy argument. It may be a variable name, array name, or dummy procedure name.

A FUNCTION statement must appear only as the first statement of a function subprogram.

The name of an external function is a global name and must not be the same as any other global name or any local name, except a variable name, in the function subprogram.

The name of the function subprogram must appear as a variable name in the subprogram. During every execution of the external function, this variable must become defined, and once defined, may be referenced or become redefined. The value of this variable when a RETURN or END statement is executed in the subprogram is the value of the function. If this variable is a character variable with a length specification that is an asterisk in parentheses, it must not appear as an operand for concatenation except in a character assignment statement. In addition to returning the value of the function, an external function may define dummy arguments to return values.

If the type of a function is specified in a FUNCTION statement, the function name must not appear in a type statement. A name must not have its type explicitly specified more than once in a program unit.

When an external function is referenced in a program unit, it must be of the same type as the function declaration. If a function is a character function, the length of the character function in the program unit in which it is referenced must be the same as the declared length.

In a function subprogram, the name of a dummy argument is local to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name. A character dummy argument whose length specification is an asterisk in parentheses must not appear as an operand for concatenation, except in a character assignment statement.

The actual arguments in an external function reference must agree in order, number, and type with the corresponding dummy arguments in the referenced function. The use of a subroutine name as an actual argument is an exception to the rule requiring agreement of type.

The actual arguments in the reference of an external function may be an array name, an intrinsic function name, an external procedure name, a dummy procedure name, or any expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is the symbolic name of a constant.

Note that if there are no dummy arguments in the function statement, the function must be referenced with an empty actual argument list.

Following are sample function statements and sample references to these functions:

```
FUNCTION ZTEMPP(I,X,Y)
DIMENSION Y(10)
...
RETURN
END
```

The preceding function may be referenced as in the following program excerpt:

```
DIMENSION A(10),B1(5)
...
...
B1(I*3) = ZTEMPP(J,C1,A)/FRT + OL(6+TOL)
...
END
```

Following is the logical function LFIND:

```
LOGICAL LFIND()
...
RETURN
END
```

This function may be referenced as in the following program excerpt:

```
LOGICAL LFIND
...
...
IF (LFIND()) GO TO 95
....
```

Following is the character function RANGER:

```
CHARACTER RANGER(A,B)
CHARACTER A,B*(*)
...
...
RETURN
END
```

This function may be referenced as in the following program excerpt:

```
CHARACTER RANGER,CHR1,CHR2,CHR3
...
...
IF (RANGER(CHR1,CHR2).LT.CHR3) GO TO 35
...
```

Following is the character function CCONVT:

```
CHARACTER*2  CCONVT(A)
CHARACTER A*(*)
...
...
RETURN
END
```

The preceding function may be referenced as in the following program excerpt:

```
CHARACTER CCONVT,CHAR1*2,CHAR2*6,CHAR3*8
...
CHAR3 = CCONVT(CHAR1)//CHAR2
...
...
```

SUBROUTINE

A subroutine is a procedure that is specified external to the program unit that references it. It is specified in a subroutine subprogram.

A subroutine subprogram is a sequence of statements initiated by a subroutine statement and terminated by an END statement. The SUBROUTINE statement indicates the beginning of a subroutine subprogram and may specify the dummy arguments employed in that subprogram.

The format of a SUBROUTINE statement is:

```
SUBROUTINE s  [( [d [,d]...])]
```

where: s is the symbolic name of the subroutine subprogram in which the SUBROUTINE statement appears. s is a subroutine name.
d is a dummy argument. It can be a variable name, array name, or dummy procedure name, or is an asterisk. An asterisk indicates an alternate return.

A subroutine subprogram specifies one or more subroutines. A subroutine subprogram is a program unit that has a SUBROUTINE statement as its first statement.

The name of a subroutine is a global name and must not be the same as any other global name or any local name in the program unit.

In a subroutine subprogram, the name of a dummy argument is local to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name. A character dummy argument whose length specification is an asterisk in parentheses must not appear as an operand for concatenation, except in a character assignment statement.

A subroutine subprogram is referenced by a CALL statement. Note that if there are no dummy arguments, either of the forms s or s() may be used in the SUBROUTINE statement. A subroutine specified by either form may be referenced by a CALL statement of the form CALL s or CALL s().

Following are examples of subroutine statements:

```
SUBROUTINE SUBA
SUBROUTINE MULTMT(ARRAY1,ARRAY2)
SUBROUTINE MAXI(A,B,C,*,D,E)
SUBROUTINE GFOR(Z,FUNC)
SUBROUTINE PHASEA()
```

CALL STATEMENT

A subroutine is referenced by a CALL statement.

The format of a CALL statement is:

```
CALL s [( [a [,a]...])]
```

where: s is the name of a subroutine or dummy procedure, and
a is an actual argument.

Execution of a CALL statement results in the evaluation of actual arguments that are expressions, the association of actual arguments with the corresponding dummy arguments, and the transfer of control into the subroutine subprogram that is referenced. Return of control from the referenced subroutine completes execution of the CALL statement.

The actual arguments in a subroutine reference must agree in order, number, and type with the list of arguments in the referenced subroutine. The use of a subroutine name or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type.

The actual arguments in a CALL statement must be an array name, an intrinsic function name, an external procedure name, a dummy procedure function name, an alternate return, dummy procedure, an alternate return specifier, or an expression. A character expression involving concatenation of an operand whose length specification is an asterisk in parentheses is not used, unless the operand is the symbolic name of a constant.

An alternate return specifier has the form *l where l is a statement label of an executable statement that appears in the same program unit as the CALL statement.

Following are examples of CALL statements:

```
CALL SUBA()
CALL MULTMT(ARRAYA,ARRAYB)
CALL MAXI(U,V,W,*30,X,Y)
CALL GFOR(2*C(3)/4.1,FACTOR)
CALL PHASEA
```

ENTRY STATEMENT

An ENTRY statement permits a procedure reference to begin with a particular executable statement within the function or subroutine subprogram in which the ENTRY statement appears. One or more ENTRY statements may appear within a function subprogram after the FUNCTION statement, or within a subroutine subprogram after the SUBROUTINE statement. An ENTRY statement must not appear between a block IF statement and its corresponding END IF statement, or between a DO statement and the terminal statement of its DO-loop.

An ENTRY statement is classified as a nonexecutable statement.

The format of an ENTRY statement is:

```
ENTRY E [( [d [,d]...])]
```

where: E is the symbolic name of an entry in a function or subroutine subprogram and is called an entry name.

d is a dummy argument. It is a variable name, array name, dummy procedure name, or is an asterisk. An asterisk indicates an alternate return and is permitted in a ENTRY statement only in a subroutine subprogram.

If an ENTRY statement appears in a subroutine subprogram, the ENTRY name is also a subroutine name and is referenced as a subroutine. If the ENTRY name appears in a function subprogram, the ENTRY name is also an external function name and is referenced as a function.

Note that if there are no dummy arguments, either of the forms E or E() may be used in the ENTRY statement. A function that is specified by either form must be referenced by the form E(). A subroutine that is specified by either form may be referenced by a CALL statement of either form.

An ENTRY statement appearing in the body of a subprogram provides an alternate point at which to begin execution when the procedure is called. Execution begins with the first executable statement after the ENTRY statement.

The dummy argument list in an ENTRY statement is independent of any other dummy argument lists in the program unit. If the same names appear in more than one dummy argument list in a subprogram, they need not occupy the same positions in those lists. An ENTRY statement may contain more or fewer dummy arguments than the FUNCTION or SUBROUTINE statement. Entry into a procedure at any point results in initialization of only those dummy arguments included in the dummy argument list of the statement in which entry occurs.

If the entry name within a function subprogram is to have a non-default type, it must be referenced in a type or IMPLICIT statement both in the calling program unit and in the function subprogram body.

Within a function subprogram all variables whose names are also the names of entries are associated with each other and with the variable, if any whose name is also the name of the function subprogram. Therefore, any such variable that becomes defined causes all associated variables of the same type to become defined and all associated variables of different types to become undefined. Such variables are not required to be of the same type unless the type is character, but the variable whose name is used to reference the function must be in a defined state when the RETURN or END statement is executed. An associated variable of a different type must not become defined during the execution of the function reference.

If an entry name in a function subprogram is of type character, each entry name and the name of the function subprogram must be of type character and have the same length specification. In a function subprogram, a variable name that is the same as an entry name must not appear in any executable statement that precedes that entry name in an ENTRY statement.

The following is an example of a subroutine containing an ENTRY statement:

```
SUBROUTINE SUBA(A,B)
A=A/2.0
ENTRY XSUBA(B,A)
A=A**B
B=A
RETURN
END
```

This subroutine may be called by statements of both these forms:

```
CALL SUBA(G,H)
CALL XSUBA(G,H)
```

The following is an example of a function subprogram containing an ENTRY statement:

```
FUNCTION QSIN(X)
A=2.3
ENTRY RSIN(X)
QSIN=1.1 + A + SIN(X)*2
RETURN
END
```

This function may be referenced, for example, in either of the following ways:

```
IF(QSIN(B).LT.1.5)GO TO 7
IF(RSIN(B).LT.1.5)GO TO 7
```

RETURN STATEMENT

The executable RETURN statement may appear only in a function or subroutine subprogram. It causes control to be returned to the referencing program unit.

The format of a RETURN statement in a function subprogram is:

```
RETURN
```

The format of a RETURN statement in a subroutine subprogram is:

```
RETURN [n]
```

where n is an integer expression.

Execution of a RETURN statement in a function or subroutine subprogram causes the return of control to the currently referencing program unit. The RETURN statement is an optional statement. If an END statement is encountered before a RETURN statement is executed the END has the same effect as executing a RETURN statement. For a function, the value of the function must be defined prior to the return and is available to the referencing program unit.

Execution of a RETURN statement (or END statement) causes all entities within a subprogram to become undefined, except for entities in blank common, entities specified by SAVE statements, entities that have been initially defined and have not been redefined, or entities in named common blocks that appear in the subprogram and appear in at least one other program unit that is referencing the subprogram, either directly or indirectly.

Note that if a named common block appears in the main program, the entities in the named common block do not become undefined at the execution of any RETURN statement in the executable program.

ALTERNATE RETURN

If the integer expression, n, is specified in the RETURN statement of a subroutine, execution of this return transfers control to a specified labeled executable statement in the calling program unit. This return statement is referred to as an alternate return.

When an alternate return is executed, the integer expression is evaluated. The resulting value, n, is used to select the nth asterisk in the dummy argument list of the procedure. If n is less than one or greater than the number of asterisks in the dummy argument list, control returns to the CALL statement that initiated the procedure reference. Otherwise, control is returned to the statement identified by the alternate return specifier in the CALL statement corresponding to the nth asterisk in the dummy argument list of the currently referenced name. This completes the execution of the CALL statement.

An alternate return specifier is of the form *s, were s is the statement label of an executable statement that appears in the same program unit as the call statement.

The following program excerpt illustrates a calling program unit able to handle alternate returns from the called subprogram.

```
      CALL SUBA(I,*33,B,*20,J)
   15 Z=A1 + N
   20 Z=Z + A2
   33 Z=Z + A1
```

The called subroutine could be the following:

```
      SUBROUTINE SUBA (I,*,B,*,J)
      IF (I) 5,6,7
    5 RETURN 1
    6 RETURN I+2
    7 RETURN
      END
```

First, suppose that when the IF statement in the subroutine is executed, I is greater than zero. Then there is a transfer of control to the statement labeled 7, which is a standard return. The subroutine returns to the calling program unit at the CALL statement, then execution continues to the statement labeled 15.

Second, suppose that I is less than zero. When the IF statement in the subroutine is executed, there is a transfer of control to the statement labeled 5. This is an alternate return to the first label in the actual argument list. Return will be to the statement labeled 33 in the calling program unit.

If I is equal to zero when the IF statement is executed, there will be a transfer of control to the statement labeled 6. The arithmetic expression evaluates to 2, causing a return to the statement labeled 20 in the calling program unit.

If a RETURN 3 or RETURN 0 had been attempted in SUBA, a standard return would have been executed.

ARGUMENTS AND COMMON BLOCKS

Arguments and common blocks provide a means of communication between the referencing program unit and the referenced procedure. The entities that appear in the argument list of the procedure are called dummy arguments; the entities that appear in the argument list of the procedure reference are called actual arguments.

Data may be communicated to a statement function or intrinsic function by an argument list. Data may be communicated to and from an external procedure by an argument list or common blocks. Procedure names may be communicated to an external procedure only by an argument list.

Statement functions, external functions, and subroutines use dummy arguments to indicate the type and whether each argument is a single value, array of values, procedure, or statement label. Note that a statement function dummy argument may be only a variable, and an actual argument must not be the name of a statement function in the program unit containing the reference.

Each dummy argument is classified as a variable, array, dummy procedure, or asterisk. At the execution of a function or subroutine reference, an association is established between the corresponding dummy and actual arguments. The first dummy argument becomes associated with the first actual argument; the second dummy argument becomes associated with the second actual argument, and so on. Actual arguments may be constants, symbolic names of constants, function references, expressions involving operators, and expressions enclosed in parentheses if and only if the associated dummy argument is a variable that is not defined during execution of the referenced external procedure. Just prior to the association of dummy arguments to actual arguments, actual arguments consisting of expressions or subscripts are evaluated.

A dummy argument that is a variable may be associated with an actual argument that is a variable, array element, substring, or expression. If the actual argument is a variable name, array element name, or substring name, the associated dummy argument may be defined or redefined within the subprogram. If the actual argument is a constant, a constant reference, or an expression, the associated dummy argument must not be redefined within the procedure.

If a dummy argument is of type character, the associated actual argument must be of type character and the length of the dummy argument must be less than or equal to the length of the actual argument. If the dummy argument length is less than the length of the actual argument, the left-most characters are associated with the dummy argument.

A dummy argument that is an array may be associated with an actual argument that is an array, array element, or array element substring. In the latter two cases, the actual argument may be thought of as an array where the first element is that array element or array element substring. Care should be taken that a dummy array is not declared larger than the corresponding actual array, or an invalid index may result when the dummy array is referenced.

When considering the length of a character type array, the overall character length of the array should be considered, rather than the number of elements in the array. The length of an element in the dummy argument array may be different from the length of an array element in the actual array. The character array should be thought of as a string of characters.

A dummy argument that is an asterisk may be associated only with an actual argument that is an alternate return specifier in the CALL statement.

A dummy argument that is a dummy procedure may be associated only with an actual argument that is an intrinsic function, external function, subroutine, or another dummy procedure.

The variables and arrays in a common block may be defined and referenced in all subprograms that contain a declaration of that common block. Because association is by storage rather than by name, the names of the variables and arrays may be different in the different subprograms.

SECTION 12

BLOCK DATA SUBPROGRAM

INTRODUCTION

Block data subprograms are used to provide initial values for variables and array elements in common blocks. A block data subprogram is non-executable.

A block data subprogram is a program unit that has a BLOCK DATA statement as its first statement. There may be more than one block data subprogram in an executable program.

The format of a BLOCK DATA statement is:

```
BLOCK DATA [s]
```

where s is the optional name assigned the block data subprogram in which the BLOCK DATA statement appears.

The optional name is a global name and must not be the same as any local name in the subprogram, the name of an external procedure, main program, common block, or other block data subprogram in the same executable program.

The BLOCK DATA statement must appear only as the first statement of a block data subprogram. The only other statements that may appear in a block data subprogram are IMPLICIT, PARAMETER, DIMENSION, COMMON, SAVE, EQUIVALENCE, DATA, END, and TYPE statements. Comment lines are also permitted.

If an entity in a common block is initially defined, all entities having storage units in the common block storage sequence must be specified even if they are not all initially defined. More than one common block may have entities initially defined in a single block data subprogram.

The same named common block may not be specified in more than one block data subprogram in the same executable program. There must not be more than one unnamed block data subprogram in an executable program.

The following is an example of a block data subprogram:

```
BLOCK DATA
IMPLICIT LOGICAL (L)
COMMON L1, LCTL, /BLKA/A(3)
DATA L1, LCTL, A/2*.TRUE.,3*3.2/
END
```

SECTION 13

STORAGE ALLOCATION

INTRODUCTION

Each B 6000/B 7000 series FORTRAN 77 data item is allocated one or more units of storage depending upon the type of value the item represents. The unit of storage is the data word, 48 bits of which are accessible to the user.

The following FORTRAN 77 features require storage configurations:

Simple Variables
Arrays
Equivalenced Data Items
Elements of Common Storage

The method of storage allocation for each of these features is discussed in this section. The notation [m:n] is used to describe data word fields. The 48 accessible bits of a data word are considered to be numbered, with the left-most bit being bit 47 and the right-most bit being bit 0. In the notation [m:n], m denotes the number of the left-most bit of the field being described, and n denotes the number of bits in the field. For example, the field shown (bit 28 through 24) would be described by [28:5]:

47	43	39	35	31	27	23	19	15	11	7	3
46	42	38	34	30	26	22	18	14	10	6	2
45	41	37	33	29	25	21	17	13	9	5	1
44	40	36	32	28	24	20	16	12	8	4	0

Hexadecimal constants are used extensively in the present section to describe word contents. Hexadecimal constants can be visualized as columns in the preceding diagram of a data word.

VARIABLES

Each variable requires one or two words of storage depending upon the type of variable. The variable types are:

Integer variables
Real variables
Double precision variables
Complex variables
Logical variables
Character variables

Each of these variable types has a unique internal storage configuration.

INTEGER VARIABLES

An integer variable requires one word of storage. The data word corresponding to an integer variable is partitioned as follows:

Field	Contents
[47:1]	Unused
[46:1]	Sign bit
[45:7]	Contains all 0's
[38:39]	Magnitude of data item

Integer values are represented internally in signed-magnitude notation. The sign of the value is denoted by bit 46 of the data word involved. This bit is 0 for positive values and 1 for negative values. The magnitude of the value is stored right-justified in bits 38 through 0 preceded by zeros.

For example, the internal representation of the integer 10 described using hexadecimal constant notation is :

```
Z00000000000A
```

The internal representation of -10 is:

```
Z40000000000A
```

The largest integer value that may be stored is Z007FFFFFFFFF = 549755813887.

REAL VARIABLES

A real variable requires one word of storage. The data word corresponding to a real variable is partitioned as follows:

Field	Contents
[47:1]	Unused
[46:1]	Mantissa sign bit
[45:1]	Exponent sign bit
[44:6]	Magnitude of data item exponent
[38:39]	Magnitude of data item mantissa

Real (floating point) values are represented internally in signed **magnitude** mantissa and exponent notation. The sign of the mantissa is denoted by bit 46 while the sign of the exponent is denoted by bit 45 of the data word. A minus sign is denoted by a 1 in the appropriate bit. The magnitude of the exponent is contained in bits 44 through 39; hence, the exponent may not exceed a magnitude of 2**6 - 1 = 63. The magnitude of the mantissa is contained in bits 38 through 0. This field is identical to the magnitude field of a data word contained in an integer value. Thus, an integer data word may be considered a real data word with an exponent field containing all zeros.

The value represented by a real data word is:

(mantissa) * 8**(exponent)

The magnitude of the mantissa is usually stored left-normalized within its word field. Therefore, the internal representation of the real value 0.5 (or .5E0) in hexadecimal constant notation is Z26C000000000. The exponent in this example is -13 and the mantissa is 4*8**32.

DOUBLE PRECISION VARIABLES

A double precision variable requires two adjacent words of storage. The data words corresponding to a double precision variable are partitioned as follows:

Field	Contents
First Word	
[47:1]	Unused
[46:1]	Mantissa sign bit
[45:1]	Exponent sign bit
[44:6]	Least significant part of magnitude of exponent
[38:39]	Most significant part of magnitude of mantissa
Second Word	
[47:9]	Most significant part of magnitude of exponent
[38:39]	Least significant part of magnitude of mantissa

Double precision values are represented internally in signed magnitude mantissa and exponent notation with a most significant part and a least significant part. The sign of the mantissa is contained in bit 46 of the first data word while the sign of the exponent is contained in bit 45 of the first word. A minus sign is denoted by a one in the appropriate bit. The magnitude of the exponent is contained in a total of 15 bits. The first nine of these 15 bits (the most significant part of the magnitude of the exponent) are contained in bits 47 through 39 of the second data word. The remaining six of these 15 bits (the least significant part of the magnitude of the exponent) are contained in bits 44 through 39 of the first data word.

The magnitude of the mantissa is contained in a total of 78 bits. The first 39 bits (the most significant part of the magnitude of the mantissa) are contained in bits 38 through 0 of the first data word. The remaining 39 bits (the least significant part of the magnitude of the mantissa) are contained in bits 38 through 0 of the second word.

The value represented by a double precision data-word pair may be obtained by the formula:

```
((most significant part of mantissa) + (least significant part of
mantissa)*8**(-13)) * 8**((most significant part of exponent) +
(least significant part of exponent) *2**9))
```

The magnitude of the mantissa is usually stored left-normalized within the total 78 bit field.

As an example, the internal two-word representation of the double precision value 1D0 in hexadecimal constant notation is:

```
Z261000000000 Z000000000000
```

The contents of the first of these two data words are identical to the contents of a data word representing the real value 1E0. When a value which does not exceed the limits of a real variable is assigned to a double precision data word pair, then that data word pair can be represented in a form where the second data word is all zeros and the first data word is identical to the real representation of the value.

COMPLEX VARIABLES

A complex variable requires two adjacent words of storage. The first data word contains the real part of the variable, and the second data word contains the imaginary part of the variable.

Each of these two data words conforms to the format of an integer or real variable. If the complex variable is assigned the value of a complex constant whose real and/or imaginary part is an integer constant, then the first and/or second data word of the complex constant will be partitioned as an integer data word. A real constant in the complex constant similarly controls the structure of the corresponding complex data word.

For example, if C is complex, the assignment statement:

```
C = (1,0.5E0)
```

places into the two data words corresponding to C these values:

```
Z000000000001 Z26C000000000
```

The first data word (the real part of C) contains a representation of the integer constant 1. The second data word (the imaginary part of C) contains a representation of the real constant 0.5E0. If the assignment statement were:

```
C = (1,1)
```

then the data words would be:

```
Z000000000001 Z000000000001
```

LOGICAL VARIABLES

A logical variable requires one word of storage. The data word corresponding to a logical variable is partitioned as follows:

Field	Contents
[47:47]	Unused
[0:1]	Value bit

The left-most 47 accessible bits of the data word containing the value of a logical variable has no effect on the value, while the value represented by that variable is contained in bit 0 of the data word. When bit 0 is 1, the value of the variable is .TRUE. and when bit 0 is 0, the value of the variable is .FALSE..

Usually, the internal representation of a logical value is identical to the representation of an integer 1 or integer 0. However, since only the state of bit 0 is taken into consideration, the remaining 47 bits may assume any states. Logical operators reference all 48 bits.

CHARACTER VARIABLES

Character variables are stored as strings of EBCDIC characters. Character variables not in common are collected and stored together in character pools. These character pools are stored internally as EBCDIC arrays.

ARRAYS

FORTRAN arrays are provided to allow the user to organize program storage locations into a convenient structure. Internally, an array is stored as a group of one or more contiguous storage units. A FORTRAN array of any number of declared dimensions is represented internally by a one-dimensioned array of storage locations. Each element of an array has storage requirements and partitioning identical to that of a variable of the same type as the array. Thus, each element of a real array requires one data word partitioned like a real variable and each element of a complex or double precision array requires two data words partitioned like a complex or double precision variable, and each element of a character array requires the number of characters specified in the length specification for the array. If the length is not specified the length is one character.

Each integer, real, and logical array is allocated as many data words as there are elements in the array. Each double precision and complex array is allocated twice as many internal data words as there are elements in the array. Each character array is allocated as a string of EBCDIC characters. The total number of characters in the array is equal to the number of elements in the array times the length of the array elements. Each character array is stored in its own character pool which is stored internally as an EBCDIC array.

The elements of an array are ordered into a sequence. The subscript of an array element has a subscript value which determines which element of the sequence is identified by the array element. The first element has a subscript value of 1.

If an array is declared by the array declarator:

```
A(L1:U1,L2:U2,L3:U3,...,Ln:Un)
```

the subscript value for the array element:

```
A(S1,S2,S3,...Sn)
```

would be

```
1+(S1-L1)
 + D1*(S2-L2)
 + D1*D2(S3-L3)
 + ...
 + D1*D2*...*D(n-1)*(Sn-Ln)
```

where: Dk is the size of the kth dimension, equal to Uk-Lk+1, where Uk and Lk are integers,
n is the number of dimension, and
Sk is the kth subscript expression, which is an integer.

The offset of an array element within the sequence is its position in the sequence relative to the first element. The offset of an array element is equal to its subscript value minus 1. The offset of the first element of an array is 0.

Example:

For an array declared:

```
DIMENSION A(0:5,1:4)
```

the subscript value for A(3,2) would be:

```
= 1 + (3-0) + 6*(2-1)
= 1 + 3 + 6
= 10
```

The ordering of the array elements produced by the subcript value is identical to that obtained by listing all of the array elements by varying the first subscript expression most rapidly, the second subscript expression next most rapidly, and so on.

Example:

For the array A declared by the declarator:

```
DIMENSION A(0:2,1:2,-2:-1)
```

the elements are stored in the following order:

```
A(0,1,-2)
A(1,1,-2)
A(2,1,-2)
A(0,2,-2)
A(1,2,-2)
A(2,2,-2)
A(0,1,-1)
A(1,1,-1)
A(2,1,-1)
A(0,2,-1)
A(1,2,-1)
A(2,2,-1)
```

Element A(0,1,-2) corresponds to the first element of the internal array. This order is the order in which array elements are considered when a multidimensional array is used in an EQUIVALENCE statement or when the name of a multidimensional array appears in an I/O list, DATA statement, argument list, COMMON statement, as a format designator, or as an internal file identifier.

An array is stored internally in consecutive locations as long as the storage requirement of the array does not exceed 1023 words. Arrays exceeding this size are segmented for overlay purposes unless the compiler option LONG is set. Each segment contains a maximum of 256 words and is automatically overlayed and recalled as necessary during execution of the program.

EQUIVALENCED DATA ITEMS

The EQUIVALENCE statement allows the user to assign a number of program data items to a single storage unit. Thus, more than one variable name or array element name may refer to one storage location.

SINGLE STORAGE LOCATIONS

The least complicated use of the EQUIVALENCE statement involves the assignment of data items requiring a single word of storage to a mutual storage location. As an example, assume that the following statements are the first statements of an executable program:

```
IMPLICIT INTEGER (A), REAL (B), LOGICAL (L)
DIMENSION AR(2)
EQUIVALENCE (A, AR(2),B2), (AL,L)
```

The EQUIVALENCE statement causes the integer variable A, the integer array element AR(2), and the real variable B2 to all be assigned to the same data word. The first element of AR is not affected by this statement. A change in the value of any one of the three equivalenced items produces a simultaneous, equivalent change in the value of the other two items.

The EQUIVALENCE statement also causes the integer variable AL and the logical variable L to be assigned to the same data word. As the variable L changes value, AL also changes value. For example, the assignment statement:

```
L = .TRUE.
```

places the integer value 1 into variable AL. Also, as the variable AL changes value, L changes value accordingly. As AL assumes even integer values, L assumes the value .FALSE., and as AL assumes odd integer values, L assumes the value .TRUE..

MULTIPLE STORAGE LOCATIONS

EQUIVALENCE statements may also involve data items requiring more than one word of storage. As an example, assume that the following statements are the first statements of a program:

```
DOUBLE PRECISION D
REAL A(2)
EQUIVALENCE (A,D,B)
```

The EQUIVALENCE statement causes the real array A, the double precision variable D, and the real variable B to be assigned to identical data words. As A and D both require two data words, the first and second elements of A become equivalent to the first and second words, respectively, of the storage unit assigned to D. The variable B, requiring only one data word, is assigned to the same word as A(1) and the first word of D.

ARRAY HANDLING

The EQUIVALENCE statement may be used to assign a group of contiguous storage locations to a number of program arrays. There are two types of possible array references in an EQUIVALENCE statement: an array name and a subscripted array name with the same number of subscript expressions as the array has dimensions.

Assume the following statements are the first statements of a program unit:

```
REAL A(4), B(10,10), C(100), D(50), E(3,3), F(50)
DOUBLE PRECISION DE(2) /2*3D0/
EQUIVALENCE (A,DE), (B(1,1), C(1)), (D,F(26), E(2,1))
```

The first list in this EQUIVALENCE statement causes the real array A and the double precision array DE to share four storage words. The first two elements of A become equivalent to the two words of the first element of DE, and the last two elements of A become equivalent to the two words of the second element of DE. The initial value list in the type statement declaring DE causes the double precision value 1D0 to be placed in DE(1) and DE(2) and in the word pairs A(1), A(2) and A(3), A(4).

The second list in the EQUIVALENCE statement causes the real arrays B and C to share 100 storage words. Each element in the 100 element array C, beginning with C(1), is assigned to the same storage unit as a unique element of B. The elements of the two-dimensional array B are stored internally in the manner described in the discussion of arrays in this section. The internal storage units assigned to C occur in the same order as the elements of that array. Hence, each C(I) is equivalenced to the Ith internal element of B and it follows that if the following two PRINT statements occur in this program unit, identical output is produced:

```
PRINT 1, (C(I), I=1,100)
PRINT 1, B
```

The third list in the sample EQUIVALENCE statement indicates that the elements of the arrays D, F, and E are to be equivalenced in such a manner that D(1), F(26), and the second internal element of the two-dimensional array E are to be assigned identical internal storage units. Therefore, the elements F(26) through F(50) become equivalent to the first 25 elements of D. Equivalencing is handled in the manner illustrated. Each of the lines denotes a single storage location, and the array elements on a line are assigned to the same storage location.

	F(1)	
	through	
	F(25)	E(1,1)
D(1)	F(26)	E(2,1)
D(2)	F(27)	E(3,1)
D(3)	F(28)	E(1,2)
D(4)	F(29)	E(2,2)
D(5)	F(30)	E(3,2)
D(6)	F(31)	E(1,3)
D(7)	F(32)	E(2,3)
D(8)	F(33)	E(3,3)
D(9)	F(34)	
through	through	
D(25)	F(50)	
D(26)		
through		
D(50)		

The following DATA statement would initialize the elements E(1,2), E(2,2), F(28), F(29), D(3), and D(4) with the value 6:

```
DATA (E(J,2), J=1,2) / 2*6./
```

ELEMENTS IN COMMON STORAGE

The COMMON statement allows program values to be communicated between program units without employing argument lists and permits these data items to be referenced by the same or different symbolic names in each program unit. This statement also allows data initialization via a BLOCK DATA program unit.

Each element of a common block is allocated storage in common storage once for an executable program. Each program unit may reference a common block (and hence each location in the block) through an appropriate COMMON statement.

The size of each block of common storage is either as large as the maximum size indicated by a COMMON statement referencing that block in any program unit or as large as the maximum length to which the block is extended by an EQUIVALENCE statement.

Assume that the following statements are the first statements of a program unit:

```
      SUBROUTINE MSG
      DOUBLE PRECISION D
      LOGICAL FLAG(6)
      COMMON WORD1, WORD2,D,FLAG,TEXT(20)
      COUNT = 1
```

The COMMON statement is assumed to provide the maximum size of the blank common block in the executable program containing program unit MSG. The total size of this common block is then 30 words. These words are denoted in the MSG subprogram as the words assigned to the real variables WORD1 and WORD2, the word pair assigned to the double precision variable D, the six words assigned to the logical array FLAG, and the twenty words assigned to the real array TEXT. These data words are contained within the common block in the order listed.

The blank common block just described could be referenced, for example, by a common statement within another program unit:

```
      SUBROUTINE DUMP
      COMMON T(10)
      PRINT 1,T
    1 FORMAT(X,Z12)
      RETURN
      END
```

The elements of real array T would therefore correspond to the first 10 data words contained in the common block. The last 20 data words of the common block are not accessed in the DUMP subprogram.

Additionally, the EQUIVALENCE statement may interact with the COMMON statement to extend the length of a common block beyond the maximum size specified for the block by COMMON statements.

As an example, assume that the following statements form two units of an executable program:

```
FUNCTION SUM(N)
COMMON /GR1/IT(3,3)
.
.
.
RETURN
END

LOGICAL FUNCTION TEST(L)
LOGICAL X(6)
COMMON /GR1/K(9)
EQUIVALENCE (K(6),X)
.
.
.
RETURN
END
```

The common block referenced by these two program units is named GR1 and consists of 11 data words. The function SUM accesses the first nine words of this block through the two-dimensional integer array IT. The function TEST accesses the first nine words of the block through the integer array K. In addition, the next two locations of GR1 are referenced as the logical array elements X(5) and X(6) since X is equivalenced to K starting at the element K(6).

Array K and array IT both refer to the same area of common storage. Use of array K allows these locations to be referenced using only one subscript expression. Equivalencing portions of the X and K arrays allow some elements of GR1 to be handled as both integer and logical type items.

The common block could not have been extended backwards by the EQUIVALENCE statement. The following combination of statements is invalid:

```
LOGICAL X(6)
COMMON /GR1/K(9)
EQUIVALENCE (X(3),K)
```

SECTION 14

COMPILER INFORMATION

INTRODUCTION

The B 6000/B 7000 series FORTRAN 77 compiler is written in B 6000/B 7000 Extended ALGOL. This program compiles source programs producing object programs which may be executed on the B 6000/B 7000 system. The FORTRAN 77 compiler and the object programs generated by this compiler are designed to operate under control of the B 6000/B 7000 Master Control Program (MCP).

COMPILER FILES

INPUT FILES

The primary compiler input file is a file with the internal name CARD; the secondary input file is a file with the internal name SOURCE. The presence of the primary file CARD is required for each compilation; the presence of the secondary file SOURCE is optional. The file SOURCE is used as input only when the MERGE compiler option is set in the primary file. Both the CARD file and the SOURCE file may be label-equated via FILE system control cards to change their file attributes.

Additional files may be input to this compiler through use of the INCLUDE compiler control statement.

OUTPUT FILES

Output files produced by this compiler include the object code file, an optional updated symbolic file, a listing, and an error message file.

The object code file has the internal name CODE and is saved after the compilation unless the COMPILE system control card specifies compilation for syntax only, or syntax errors are detected in the source language input. If compile-and-go is specified by the COMPILE card, then the object code file is discarded after the code is executed.

The TITLE of the saved code file is identical to the program name appearing on the COMPILE card.

The updated symbolic file is generated only if the compiler option NEW is set. This file contains the compilation source input or a selected portion of this input as specified by the states of the NEW and INCLNEW compiler options and may be used for succeeding compilation. This output file has the internal name NEWSOURCE and the TITLE NEWSOURCE.

The listing is a file that is created when the compiler option LIST is set. The LIST option is set by default unless the compilation is initiated through CANDE. The file has the internal name LINE and bears the following information:

1. Compiler version.
2. Date and time of compilation.
3. Code file title.
4. Source and compiler control statements input to the compiler.
5. Code segmentation information.
6. Error messages and error count.
7. Warning messages.
8. A listing of the code generated.
9. A list of stack addresses of data items used in the program.
10. Number of input card images scanned.
11. Elapsed compilation time.
12. Processing time required for the compilation.
13. Estimated size of the program's working stack when the program is executed.
14. Estimated size of the program files and related storage.
15. Estimated core storage requirement.
16. Total number of words of object code generated.
17. Number of words of array storage required for the execution of the program.
18. Number of segments in the generated program.
19. Number of disk segments required for the code file.
20. The date the compiler was compiled.

Depending upon the setting of various compiler options, this listing may contain some or all of the items listed previously. Card images from file CARD are denoted on the listing by a "C" after the card image. Card images from the file SOURCE are denoted by a "T". Card images from an included file are denoted by an integer from 1 to 5 indicating the level of INCLUDE nesting. A "P" denotes a CARD image that has superseded a SOURCE card image.

The output error message file with the internal name and TITLE of ERRORFILE is created when the ERRORLIST compiler option is set. This file is normally used for compilations initiated through CANDE, in which case ERRORLIST is set by default and the ERRORFILE compiler file is assigned to the remote device involved. When a syntax error is detected, the offending card image is written on this file, and an error message is written on the following line of text.

COMPILER FILE TABLE

The FORTRAN77 compiler input and output files are listed in table 14-1, which provides information concerning the configuration of each file. The attributes of any of these files may be changed through the use of FILE system control card images.

INPUT DECKS

The following discussion applies to FORTRAN77 compilations that are not initiated through CANDE.

For a compilation, the only required card images are:

1. A BEGIN JOB system control card image.

2. A COMPILE system control card image to initiate the compilation and to specify the type of compilation to be performed.

3. An END JOB system control card image.

System control card images may be placed after the COMPILE card image to specify compilation parameters and execution parameters, and to alter the attributes of files (see Work Flow Language Reference Manual, form no. 5001555).

If the deck is to contain source input, the system control card images are followed by a FORTRAN or COMPILER, EBCDIC, DATA, ASCII, or BCL system control card image, depending on the form of the source input (EBCDIC and DATA are synonymous), and this card image is followed by the source input and compiler control cards. If the deck is to contain input data, then the source card images are followed by the data decks consisting of an EBCDIC, DATA, BCL, or BINARY system control card images, bearing the TITLE of the data file, followed by the card images bearing the data. If this data is coded in BINARY format, then the data is terminated by a binary end (BEND) card.

Table 14-1. Compiler Files

Internal Name	Purpose	Kind	Code	Record Size	Block Size	Comments
CARD	Primary Input File	READER	EBCDIC BCL	14 Words 10 Words	May be Blocked or Unblocked	Required for each compilation. CANDE file is equated to this file automatically for compilations initiated through CANDE.
SOURCE	Secondary Input File	DISK	EBCDIC BCL	14 Words 10 Words	May be Blocked or Unblocked	Optional file. Selected as input by setting MERGE compiler option.
May be Specified On INCLUDE Compiler Control Card	Library Include File	DISK	EBCDIC BCL	14 Words 10 Words	May be Blocked or Unblocked	Optional input file opened by $INCLUDE card bearing appropriate file TITLE or internal file name. Five levels (maximum) of nesting permitted (see discussion of INCLUDE option).
CODE	Object Code File	DISK	Hexadecimal	30 Words	270 Words	Generated object code file. Saved or discarded and assigned a TITLE as indicated in text.
NEWSOURCE	Updated Symbolic Output File	DISK	EBCDIC	14 Words	420 Words	Optional output file produced when NEW compiler option is set. This file contains portions of the source input, is assigned a default TITLE of SOURCE, and is suitable for use in a later compilation.
LINE	Listing	PRINTER	EBCDIC	20 Words	20 Words	Optional file produced when either the compiler option LIST. Label-equatable to REMOTE.
ERRORFILE	Error Listing Output File	PRINTER	EBCDIC	14 Words	14 Words	Error listing file produced when ERRORLIST compiler option is set. Contains card images and error messages.

Four kinds of FORTRAN77 compilations are available to the B 6000/B 7000 user. They are:

Compile for Syntax
Compile for Library
Compile and Go
Compile for Library and Go

COMPILE FOR SYNTAX

Compile for syntax allows a program to be compiled for syntax-checking purposes only. The object code file is not saved after the compilation.

The form of the COMPILE system control card image for a compile for syntax is:

```
COMPILE <program name> FORTRAN77 SYNTAX;
```

COMPILE FOR LIBRARY

Compile for library indicates that the object code file resulting from a successful compilation is to be saved. Execution of this code file is no initiated.

The form of the COMPILE system control card image for a compile for library is:

```
COMPILE <program name> FORTRAN77 LIBRARY;
```

The code file resulting from a compile for library may be executed via a RUN or EXECUTE system control card. For execution, the only required card images are:

1. A BEGIN JOB system control card image.

2. A RUN or EXECUTE system control card image bearing the TITLE of the the code file to be executed.

3. An END JOB system control card image.

System control card images may be placed after the RUN or EXECUTE card image to specify execution parameters and to alter file attributes. If the deck is to contain input data, the system control card images are followed by data decks consisting of an EBCDIC, DATA, BCL, or BINARY system control card images bearing the TITLE of the data file, followed by the data card images. If this data is coded in BINARY, the data is terminated by a binary end (BEND) card.

COMPILE AND GO

Compile and go indicates that the object code file resulting from a successful compilation is to be immediately executed and then discarded.

The form of the COMPILE system control card image for a compile and go follows:

```
COMPILE <program name> FORTRAN77 [GO];
```

COMPILE FOR LIBRARY AND GO

Compile for library and go indicates that the object code file resulting from a successful compilation is to be immediately executed and saved.

The form of the COMPILE system control card image for a compile for library and go follows:

```
COMPILE <program name> FORTRAN77 LIBRARY GO;
```

COMPILATIONS THROUGH CANDE

A FORTRAN77 compilation initiated using the Command and Edit Language (CANDE) differs from other FORTRAN77 compilations in the following ways:

1. Compiler option LIST is reset by default.

2. Compiler options FREE, ERRORLIST, and LINEINFO are set by default.

3. Compiler option ERRORLIMIT has a default value of 10.

4. The ERRORFILE into which syntax errors are written is equated to the remote terminal initiating the compilation.

5. Unless otherwise specified.

   1) The CARD file will be equated to the work file or specified disk file.

   2) No SOURCE file is used.

   3) The object code file created by compiling the file <filename> is <usercode> OBJECT/<filename>.

6. All compilations are compile for library. Execution of the object code may be initiated by either of two CANDE commands: RUN or EXECUTE.

SECTION 15

COMPILER OPTIONS

INTRODUCTION

Compiler control statements control how FORTRAN 77 processes source input and handles input errors. These control statements may also affect the compiler generated output. Compiler control statements are input on compiler control card images.

Compiler control information appears in columns 1 through 72 of the control card image. An eight digit sequence number may optionally appear in columns 73 through 80. Control card images may not be continued.

Each control card image is identified by a dollar sign ($) in column 1 of the card image or a blank in column 1 and a dollar sign in column 2. In addition to the dollar sign, compiler control information may consist of compiler option actions, compiler option names, and associated option arguments. Blanks separate compiler options and option actions. A blank may optionally appear between the dollar sign and the first compiler option or option action.

A compiler option may be either a parameter option or a standard option.

Parameter options are initiated by the appearance of the option name and any associated option arguments on a compiler control card image.

Standard options may have a state of SET or RESET. Each standard option has a default state and the state may be specified on a compiler control card image. A standard option which has a default state of RESET is initially assigned a 24-bit stack filled with zeros; a standard option which has a default state of SET is initially assigned a 24 bit stack with a 1 on top and 0's in the remaining positions. The top stack position denotes the state of the option at any time.

Compiler control statements assign a desired state to a compiler option or a group of compiler options. A compiler control statement, which applies to standard options, begins with an explicit action, contains a list of standard options to which the option action is to apply, and ends when the next explicit option action is encountered on the same compiler control card image.

An explicit option is one of the mnemonics:

    SET
    RESET
    POP

If a compiler control statement begins with the option action SET, the standard options following the option action are assigned the state SET; the states of all other standard options are unchanged. If the control statement begins with the option action RESET, the standard options following the option action are assigned the state RESET; the states of all other standard options are unchanged. If the specified option action is POP, then the standard options indicated revert to their immediately previous states. (Their states become RESET if these options have not been changed from their default states.) The states of all other options are unchanged.

Any parameter option, except INCLUDE, may appear on a compiler control statement beginning with an explicit option action and the parameter option action is initiated. The INCLUDE parameter option must be the only option on the compiler control statement.

Each SET option action causes the stacks allocated to the designated standard options to be pushed down one bit with a 1 placed on top. Each RESET option action causes the appropriate stacks to be pushed down one bit with a 0 placed on top. POP causes the stacks corresponding the designated options to be POPed up one bit, causing the associated options to revert to their immediately previous states. A maximum history of 24 states may be recorded.

An option affected by a particular compiler control statement retains the indicated state for all input card images with sequence numbers greater than the sequence number on the compiler control card image, or for all physically succeeding input card images when all sequence number fields are blank, until another compiler control statement is encountered which affects the option. Within a card image options are assigned states from left to right.

It is possible to assign the state of one compiler option to the state of another compiler option; for example:

```
$ SET DOUBLE = LIST
```

The following are examples of compiler control statements employing the SET, RESET, and POP explicit option actions:

```
$ SET LIST DOUBLE NEW INCLNEW
$ RESET VOID
$ POP NEW NEWSEQERR
```

The following compiler control card image:

```
$ SET LIST SEQ 10+5
```

assigns the state SET to the options LIST, and SEQ (with the sequencing arguments of 10 and +5).

The FORTRAN77 compiler options are discussed alphabetically in the following paragraphs. The default of the LIST option is SET and the default state of the balance of the standard options is RESET unless the compiler is called from CANDE, in which case the default state of the ERRORLIST, FREE, and LINEINFO options is SET and the default state of the balance of the standard options is RESET.

ASCII (RESET)

The ASCII compiler option allows the compiler to accept source programs in ASCII code.

B 7700 (RESET)

When the B 7700 option is set the compiler generates code better suited for the B 7700.

CHECK (RESET)

The CHECK compiler option causes the compiler source input files and updated source file NEWSOURCE to be checked for sequence number errors. A sequence error condition exists when a sequence number on a source card image is less than or equal to the sequence number on the preceding card image. A warning message is placed on the output listing when such an error is found. When the CHECK, SEQERR, and NEWSEQERR options are reset, the compiler performs no sequence checking.

CODE (RESET)

The CODE compiler option causes the output listing to contain the object code generated by the compiler, LIST must be set to produce this listing.

DELETE (RESET)

The DELETE compiler option limits source input. When DELETE is set, only input from file CARD is compiled; input from the secondary file SOURCE, including compiler control cards, is ignored until DELETE is reset. The ignored input is neither listed nor included in the updated symbolic file regardless of the states of LIST and NEW.

DOUBLE (RESET)

The DOUBLE compiler option alters the format of the compilation listing. When DOUBLE is set, the listing is DOUBLE spaced; when DOUBLE is reset, the listing is single spaced.

END (NO ASSOCIATED STATES)

The END compiler option signals an end of file on the medium where it is encountered.

ERRORLIMIT (NO ASSOCIATED STATES)

The parameter compiler option ERRORLIMIT allows the user to indicate that compilation is to be terminated when a certain maximum number of syntax errors are detected in the source input.

The proper format of the ERRORLIMIT compiler control statement is:

```
ERRORLIMIT [=] n
```

where n is an unsigned integer constant specifying the maximum number of syntax errors.

A limit of 0 indicates there is no limit on the number of errors allowed. The ERRORLIMIT control statement may appear only on a compiler control card which precedes the first source input statement. The default error limit is 200 unless the compilation is initiated through CANDE, in which case the default error limit is 10.

ERRORLIST (RESET FOR NON-CANDE COMPILATIONS, SET FOR CANDE COMPILATIONS)

The ERRORLIST compiler option lists compilation errors separate from the listing of the source input listing. When set, the ERRORLIST option outputs the error listing to the compiler output file ERRORFILE. When a compilation error is detected in the source input, the offending line of text and an error message are written to ERRORFILE. This option is provided primarily for use when the FORTRAN77 compiler is called from a remote terminal by CANDE, but it may be used regardless of the manner in which the compiler is called. When the compiler is called from CANDE, ERRORFILE is automatically equated to the remote device.

FREE (RESET FOR NON-CANDE COMPILATIONS, SET FOR CANDE COMPILATIONS)

The FREE compiler option allows the compilation of free-form source programs from the source file CARD. This option allows simple input from remote devices. The following restrictions must be observed:

1. Continuation card images must contain a minus sign (-) or asterisk (*) as the first non-blank character in the first six columns.

2. Comments must have a "C" in column one, and a minus sign or asterisk in column 2.

3. For statements with labels, the label must appear in the first five columns.

4. Each free-form statement may contain 66 characters at most, including blanks and not including sequence numbers. Any characters beyond the 66th character but before column 73 are flagged with a warning and ignored.

5. Continuation card images are shifted to the right until the continuation character "*" or "-" is in column 6. If this shift causes a card image to extend past column 72 and if there are blanks between the continuation character and the source, the source is shifted to the left until either the card image no longer extends past column 72 or until the source begins in column 7.

6. Columns 73 through 80 of the source card image are reserved for the sequence numbers, as in the case of formatted source input.

7. Compiler option statements must have a dollar sign ($) in column 1 or a blank in column 1 with a dollar sign in column 2.

8. FILE statements must have the string FILEbb in columns 1-6 where b represents a blank.

FREENEW (RESET)

When FREENEW and NEW compiler options are set, the updated symbolic output file NEWSOURCE is created in a form that can be compiled with FREE set.

FREESOURCE (RESET)

The FREESOURCE option permits the source file SOURCE to be free format.

INCLNEW (RESET)

The INCLNEW compiler option controls the contents of the updated symbolic file NEWSOURCE. When INCLNEW and NEW options are set, the source text, which is input to the compiler via the INCLUDE compiler option, is output to file NEWSOURCE. The $ INCLUDE card images do not appear in NEWSOURCE. If INCLNEW is reset, then the INCLUDED text will not be placed on the NEWSOURCE file; however, $ INCLUDE cards will be output to NEWSOURCE when the $ is in column 2. If the state of the NEW option is RESET, the INCLNEW option is ignored.

INCLUDE (NO ASSOCIATED STATES)

The special action compiler option INCLUDE allows the user to input text from specified files in addition to the CARD and SOURCE files. When the compiler is accepting source input from the CARD or SOURCE file and a $ INCLUDE card image is encountered, the indicated portion of the file denoted on this card image is inserted into the input text in place of the $ INCLUDE card image. The included text may also contain $ INCLUDE card images, but only five text nesting levels are permissible.

The following is the proper form of the INCLUDE compiler control statement:

```
INCLUDE [f]  [m]  [- n]
```

where f is either an internal file name or a file TITLE enclosed in apostrophes or quotes,
m is either unsigned integer constant from one to eight digits or an asterisk, and
n is an unsigned integer constant from one to eight digits.

In the format for the INCLUDE compiler control card m indicates the sequence number of the first card image to be included from the file referenced by this INCLUDE statement. If n is an asterisk rather than a sequence number, then inclusion begins with either the first card image of the file or the card image following the last card image included from this file by a previous INCLUDE statement at the same level as the present INCLUDE statement. If item m is missing, inclusion begins at the same point indicated by the use of an asterisk. The item n denotes a one-to-eight digit unsigned integer constant which is the sequence number of the last card image to be included from the file until the end of the include file is encountered.

The item f denotes the file from which the include card images are to be drawn. If this item is missing from the INCLUDE statement, then the file to be employed is the file designated to be the last INCLUDE statement at that level. An error condition exists if an INCLUDE statement appears which does not specify a file and no previous INCLUDE statement at the same level has specified a file. Thus, the first INCLUDE compile control card at any of the five possible levels of text nesting must contain either a file TITLE or an internal file name.

A compiler control card bearing an INCLUDE compiler control statement may contain no other compiler control statements.

The following are examples of INCLUDE compiler control cards:

```
$ INCLUDE FILE8 00001000 - 09000000
$ INCLUDE
$ INCLUDE "SOURCE/XYZ" - 900
$ INCLUDE INCLFILE
```

The first example instructs the compiler to accept as input all card images from the library file with internal file name FILE8 which have sequence numbers between and including 00001000 and 09000000. The TITLE and other attributes of the FILE8 file may be specified on a FILE system control card.

The second example instructs the compiler to accept as input a portion of the file accessed by the last INCLUDE statement in this deck at this level. The card images to be included consist of all card images following the last card image accessed by the last INCLUDE statement at this level. If this example is in the same deck as the first example and no other INCLUDE statements intervene, the file to be accessed is the file with internal name FILE8 and the first card image to be included is the card image with the next sequence number after 09000000.

The third example instructs the compiler to accept as input a portion of the file with the TITLE of SOURCE/XYZ. The card images to be included consist of all card images between the last card image included from that file at this level and the card image with the sequence number immediately higher than 00000900.

The fourth example instructs the compiler to accept as input a portion of the file with the internal name INCLFILE. The card images to be included consist of all card images in the file between the last card image included from that file at this level and the end of the file.

LINEINFO (RESET FOR NON-CANDE COMPILATIONS, SET FOR CANDE COMPILATIONS)

The LINEINFO compiler option causes sequence number information and its relationship to code to be saved. If a program terminates abnormally, the sequence number of the termination point is provided.

LIST (SET FOR NON-CANDE COMPILATIONS, RESET FOR CANDE COMPILATIONS)

The LIST compiler option controls the generation of the LINE compiler file. An output listing is generated on the file LINE, only if the LIST option is set for some portion of the source input. The source statements for which the LIST option is set are placed in this listing. The LIST option may be set and reset as desired within the source input.

LISTDOLLAR (RESET)

The LISTDOLLAR option causes all compiler control statements to be included in the listing if the LIST option is set.

LONG (RESET)

The LONG compiler option alters array storage handling. Normally, arrays requiring more than 1023 words of storage are segmented into 256-word segments for overlay purposes. Arrays, which are declared in a portion of the source input for which the LONG option is set, are not segmented regardless of their storage requirements.

MAKEHOST (RESET)

The MAKEHOST compiler option instructs the compiler to collect and save certain information during the current compilation in order to facilitate a selective compilation at a later time. See option SEPCOMP.

MAP (RESET)

The MAP compiler option increases the amount of compilation information contained on the listing. When MAP is set, the listing contains the address couples of items named in the program being compiled. LIST must be set to produce the listing.

MERGE (RESET)

The MERGE compiler option allows the compiler to accept source input from more than one file. When the MERGE option is set, the compiler accepts source input from the file CARD. The primary input (from file CARD) is merged with the secondary input (from file SOURCE) on the basis of sequence numbers to produce the total source input to the compiler. If matching sequence numbers occur on card images from these two files, the primary input card image overrides the secondary input card image. When a primary control card image containing only a dollar sign is encountered, the secondary card image with the same sequence number is ignored. When MERGE is reset, only primary input is accepted by the compiler.

A TITLE for the source input file may be specified by following the MERGE option with the TITLE enclosed in parentheses.

NEW (RESET)

The NEW compiler option causes the updated symbolic output file NEWSOURCE to be generated. When the NEW option is set, the source input from the CARD and, if MERGE is set, SOURCE files are placed in the file NEWSOURCE. Text inserted into the source input by the INCLUDE option is also placed in the NEWSOURCE file only if the INCLNEW compiler option is also set. All compiler control cards in the source input whose dollar sign appears in column 2 are placed on the NEWSOURCE file when NEW is set.

The NEW option may be set and reset as desired throughout the source input.

The NEWSOURCE file is created despite the occurrence of syntax errors in the source input.

If the NEW compiler option and the FREE or FREESOURCE compiler options are set concurrently, the compiler performs a minor amount of editing on the output the NEWSOURCE file to ensure that the updated symbolic is acceptable as either free or fixed form input.

The TITLE may be specified for the NEWSOURCE file by following the NEW option with a TITLE enclosed in parentheses.

NEWSEQERR (RESET)

The NEWSEQERR compiler option allows sequence number error checking to be performed on the updated symbolic output file as the file is generated. When the NEW and NEWSEQERR options are set, the updated symbolic file NEWSOURCE is generated and the sequence numbers on the card images in this file are checked for errors. A sequence error condition exists when a sequence number on a card image is less than or equal to the sequence number on the preceding card image. When such a sequence error is detected, a warning error message is placed on the printer listing and the NEWSOURCE file is not saved.

OMIT (RESET)

When the OMIT compiler option is set, card images from both CARD and SOURCE files are ignored by the compiler. They are, however, listed and included in an updated symbolic file. On a listing the omitted statements are flagged by the word OMIT. See also the VOID option.

PAGE (NO ASSOCIATED STATES)

The parameter compiler option PAGE allows the user to partially control the format of the output listing. The appearance of the PAGE control statement within the source input causes a skip to the top of the next physical page. The output listing then continues, beginning with the PAGE control statement. No argument is associated with the PAGE option.

SEPCOMP (RESET)

SEPCOMP (Separate Compilation Facility) provides the capability to automatically recompile and link any subprogram unit or named main program unit. This permits individual program units to be selectively recompiled as the need arises, avoiding expensive recompilations of the entire program.

A request to the compiler for a separate compilation causes the primary input file (CARD) to be examined for patch input images. A determination is made of the environment, if any, that requires recompilation and that environment is processed as in a normal compile with secondary input images (SOURCE) merged as appropriate. As the compilation progresses, environment information is collected and, at the completion of a successful separate compilation, stored on the code file for use during a subsequent SEPCOMP. The new code file consists of any recompiled subprograms, any valid subprograms that INTERNAL were not recompiled, and any program units that might have been added.

USER INTERFACE

There are two compiler control images associated with the SEPCOMP facility: SEPCOMP and MAKEHOST. MAKEHOST instructs the compiler to collect and save certain information during the current compilation in order to facilitate a selective compilation at a later time. MAKEHOST may only be set before the first FORTRAN source image of a full compile, should not be set or reset after the initial setting, and may not be equated to another option. The file that MAKEHOST is being requested for must be a sequenced file.

SEPCOMP invokes the separate compilation facility. This option must appear before any FORTRAN source images of the patch (CARD) file. Other compiler control images may appear before the SEPCOMP image. SEPCOMP should not be referenced after the initial setting, nor should it be equated to another option. The default state for both SEPCOMP and MAKEHOST is RESET. When SEPCOMP is requested, the compiler automatically sets the MAKEHOST option, thus allowing repetitive SEPCOMPs.

Compiler Files

Compiler files are described in the following subparagraphs.

CARD

The CARD file, as in a normal compilation, is the primary input file. It contains the FORTRAN 77 symbolic images that will be applied against the SOURCE file to create a new executable program. The CARD file may be label equated to the compilers file CARD, or the default file may be used.

SOURCE

The SOURCE file is the FORTRAN 77 symbolic file that corresponds to the HOST file. This file may be label equated to the compilers FILE SOURCE. If no file SOURCE is specified, an attempt will be made to use the default SOURCE file. The SOURCE file provides the compiler with the FORTRAN 77 symbolic file which the patches specified in the CARD will be applied against. Therefore, it is the users responsibility to ensure that the SOURCE and HOST files are compatible.

HOST

The HOST file is the code file associated with the FORTRAN 77 symbolic file (SOURCE). It may be label equated to the compilers FILE HOST, or may be optionally specified as a string immediately following the SEPCOMP option on the compiler control image. The string form takes precedence over th label equation form. The HOST file will be used to provide environment information concerning the SOURCE file, as well as the source of all object code that is not recompiled during the current SEPCOMP. It is therefore extremely important that the user guarantee that the SOURCE and CODE files are compatible.

NEWSOURCE

If the NEW compiler option is set then the compiler file NEWSOURCE will be generated. This file will contain the merged (CARD and SOURCE) input which has been compiled during the current separate compilation. Note that this does not necessarily include all of the original symbolic; only those program units which were compiled.

RESTRICTIONS

Certain restrictions are associated with the ability to do separate compilations. They are as follows:

1. All symbolic file images must contain sequence field information consisting solely of numeric data.

2. Main programs must be named.

3. The types and numbers of arguments in a subprogram declaration may not be modified. The names of arguments in a subprogram declaration may be modified if the change does not cause the type of argument to change.

SEQ (RESET)

When the SEQ compiler option is set, new sequence numbers are assigned to the card images placed in the listing and the NEWSOURCE file. This option has effect only when LIST and NEW are set.

The default base sequence number is 00001000 and the default increment is 1000. These defaults-sequencing parameters may be overridden in the following manner. A one-to-eight digit unsigned integer appearing on a compiler control card not immediately preceded by INCLUDE or ERRORLIMIT is assumed to be a special action control statement associated with the SEQ option. This integer is used as the sequencing base when the SEQ option is set. If the integer is preceded by a plus sign, it is used as the increment size when the SEQ option is set. These sequencing arguments may appear on the same compiler control card image as the SEQ option, on a preceding control card image, or on a later control card image. The following are examples of sequencing arguments on the same control card image as the SEQ option:

```
$SET SEQ 100
$SET SEQ 20+1
$RESET CODE SET SEQ LIST +200
```

In the first example, sequencing of the LINE and NEWSOURCE files begins at the sequence number 00000100 and continues in default increments of 1000 if no other sequencing was specified on a previous compiler control card image. In the second example, sequencing begins at the sequence number 00000020 and proceeds in increments of 1. In the third example, sequencing begins at the default sequence number 00001000 if this is the first time SEQ is set and no other base sequence number was specified on a previous compiler control card image, and proceeds in increments of 200. Otherwise, sequencing begins at a sequence number 200 greater than the last sequence number assigned, or at the base sequence number assigned by a preceding control card.

The following is an example of a compiler control card specifying sequencing arguments but not setting the SEQ option:

```
$ 100 + 100
```

This compiler control card only specifies that, when SEQ is set, sequencing is to begin with the sequence number 00000100 and proceed in increments of 100. If the SEQ option is set when this control card appears, these two sequencing arguments take effect immediately.

SEQERR (RESET)

The SEQERR compiler option, when set, performs sequence number error checking on the source input files. A sequence error condition exists when a sequence number is less than or equal to the sequence number on the preceding card image. When such a sequence error is found, an error message is placed on the listing and the code file generated by the compilation is saved.

SUMMARY (RESET)

The SUMMARY compiler option generates minimal compilation listing. When the SUMMARY option is set and the LIST option is reset, the listing contains only compilation "trailer" information such as the number of cards scanned, number of errors, compilation times, and so on. This trailer information is also generated when LIST is set.

VOID (RESET)

The VOID compiler option limits source input. When VOID is set, all input from files CARD and SOURCE, except compiler control cards in the CARD file, is ignored by the compiler. The ignored input is neither listed nor included in the updated symbolic file regardless of the states of LIST and NEW. The VOID option may be reset only by a compiler control card in the CARD file.

WARNFATAL (RESET)

The WARNFATAL compiler option allows the user to alter the compiler's error-handling facility. When the WARNFATAL option is set, errors which otherwise would have generated warning messages are considered syntax errors.

WARNSUPR (RESET)

The WARNSUPR compiler option, when set, causes suppression of warning messages in the listing.

XREF (RESET)

The XREF compiler option outputs a cross reference of all identifiers used in a compiled program. The LIST compiler option need not be set to generate a cross reference listing. The identifiers are arranged according to the EBCDIC collating sequence with lists of sequence numbers of card images on which the identifier appears.

XREFFILES (RESET)

The XREFFILES compiler option causes files to be saved for INTERACTIVEXREF. These files have the following titles:

```
XREFREFS/<code file name>
XREFDECS/<code file name>
```

<code file name> is the name of the code file the compiler is generating.

Running SYSTEM/XREFANALYZER with a negative task value creates the same files as does the XREFFILES option.

APPENDIX A

SYNTAX DIAGRAMS

The diagrams in this appendix describe the syntax of the B 6000/B 7000 FORTRAN77 language. The diagrams are presented in essentially the same order as the syntax is presented in the text of this manual. Certain syntactic features are not represented in these diagrams. These include the use of blanks, the manner of writing statements on initial lines and continuation lines, context-dependent features such as data type requirements, sequence of required and optional parts, uniqueness and completeness of labels used, and restrictions on the use of statements. The text of this manual should be referenced when this type of information is required.

The diagrams are presented in the form of railroad diagrams.

A railroad diagram is a technique used to graphically represent the syntax of language elements. Traversing a railroad diagram from left to right or in the direction of the arrow heads, and adhering to the limits illustrated by bridges produces a syntactically valid statement.

Continuation from one line of a diagram to another is represented by a right arrow ">" appearing at the end of the current line and the beginning of the next line.

The syntax diagram is terminated by a vertical bar "|".

Items contained in broken brackets "< >" are syntactic variables which are further defined in the appendix, or require the user to supply the requested information.

Upper case items must appear literally. Minimum abbreviations are underlined.

Example

```
                                  |<-/3\------- , -------|
                                  |                      |
-- A RAILROAD DIAGRAM CONSISTS OF -----<bridges>----------------------->
                                    |                   |
                                    |-<loops>-----------|
                                    |                   |
                                    |-<optional items>--|
                                    |                   |
                                    |-<required items>--|

>- AND IS TERMINATED BY A VERTICAL BAR. ------------------------------>|
```

The following syntactically valid statements may be constructed from the above diagram:

A RAILROAD DIAGRAM CONSISTS OF <bridges> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <optional items> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <bridges>, <loops> AND IS TERMINATED BY A VERTICAL BAR.

A RAILROAD DIAGRAM CONSISTS OF <optional items>, <required items>, <bridges>,<loops> AND IS TERMINATED BY A VERTICAL BAR.

Railroad Conponents

<required items>

No alternate path through the railroad diagram exists for required items.

Example

```
-- REQUIRED ITEM ------------------------------------------------------->|
```

<optional items>

Items shown as a vertical list indicate that the user must make a choice of the items specified. An empty path through the list allows the <optional item> to be absent.

Example

```
-- REQUIRED ITEM ------------------------------------------------------->|
                  |                   |
                  |-<optional item-1>-|
                  |                   |
                  |-<optional item-2>-|
```

The following valid statements may be constructed from the previous diagram:

REQUIRED ITEM
REQUIRED ITEM <optional item-1>
REQUIRED ITEM <optional item-2>

<loop>

A <loop> is a recurrent path through a railroad diagram. A <loop> must be traversed in the direction of the arrow heads, and the limits specified by the bridges can not be exceeded. A <loop> has the following general format:

```
   |<- <bridge> ------ <return character> -|
   |                                       |
----<object of the loop>-------------------------------------------->|
```

Example

```
   |<-/1\-------- , -------|
   |                       |
------<optional item-1>--------------------------------------------->|
     |                    |
     |-<optional item-2>-|
```

The following statements can be constructed from the previous diagram:

```
     <optional item-1>
     <optional item-1>,<optional item-1>
     <optional item-2>,<optional item-1>
```

<bridges>

A <bridge> illustrates the minimum or maximum number of times a path may be traversed in a railroad diagram. There are two forms of <bridges>:

```
    /n\  n is an integer which specifies the maximum number of
         times the path may be traversed.
    /n*\ n is an integer which specifies the minimum number of
         times the path may be traversed.
```

Example

```
   |<-/2\---------- , ----------|
   |                            |
------<optional item-1>--------------------------------------------->|
     |                          |
     |-/1*\-<optional item-2>-|
```

The loop may be traversed a maximum of two times, however, the path for <optional item-2> must be traversed at least one time.

The following statements can be constructed from the previous diagram.

<optional item-1>,<optional item-2>
<optional item-2>,<optional item-2>,<optional item-1>
<optional item-2>

LETTER

```
---- A ---------------------------------------------------------------->|
   |          |
   |- B ---|
   |          |
   |- ... -|
   |          |
   |- Z ---|
```

DIGIT

```
---- 0 ---------------------------------------------------------------->|
   |          |
   |- 1 ---|
   |          |
   |- ... -|
   |          |
   |- 9 ---|
```

LABEL

```
   |<---------------|
   |                |
---- /6\ --<digit>----------------------------------------------------->|
```

VARIABLE NAME

SUBROUTINE NAME

FUNCTION NAME

BLOCK DATA SUBPROGRAM NAME

COMMON BLOCK NAME

CONSTANT NAME

ARRAY NAME

```
             |<-/5\---------|
             |              |
--<letter>--------------------------------------------------------------->|
              |             |
              |-<letter>-   |
              |             |
              |-<digit>--   |
```

ARRAY ELEMENT NAME

```
                          |<----------- , -----------|
                          |                          |
--<array name>-- ( ---/7\-<integer expression>--- ) ------------------>|
```

SUBSTRING NAME

```
----<variable name>-------- ( ------------------------- : ------------>
  |                       |    |                     |
  |-<array element name>- |    |-<integer expression>-|

>------------------------- ) ----------------------------------------->|
  |                       |
  |-<integer expression>- |
```

INTEGER CONSTANT

```
             |<-------------|
             |              |
------------/12\-<digit>---------------------------------------------->|
  |     |
  |- + -|
  |     |
  |- - -|
```

REAL CONSTANT

```
----<basic real constant>--------------------------------------------->|
  |                                      |
  |-<basic real constant>--<real exponent>-|
  |                                      |
  |-<Integer Constant>--<real exponent>----|
```

<real exponent>

```
-- E --<Integer Constant>--------------------------------------------->|
```

<basic real constant>

```
------------<integer part>-- . --------------------------------------->|
  |     |  |                                      |
  |- + -|  |- . --<fractional part>----------------|
  |     |  |                                      |
  |- - -|  |-<integer part>-- . --<fractional part>-|
```

<integer part>

<fractional part>

```
--<unsigned integer constant>----------------------------------------->|
```

DOUBLE PRECISION CONSTANT

```
----<basic real constant>---<double precision exponent>--------------->|
  |                         |
  |-<Integer Constant>----|
```

<double precision exponent>

```
-- D --<Integer Constant>--------------------------------------------->|
```

COMPLEX CONSTANT

```
-- ( ---<real constant>------ , ---<real constant>------ ) ----------->|
       |                    |     |                    |
       |-<integer constant>-|     |-<integer constant>-|
```

LOGICAL CONSTANT

```
---- .TRUE. ---------------------------------------------------------->|
  |          |
  |- .FALSE. -|
```

HEXADECIMAL CONSTANT

```
       |<-----------------------|
       |                        |
-- Z ---/24\-<hexadecimal digit>-------------------------------------->|
```

```
<hexadecimal digit>

---- 0 ------------------------------------------------------------->|
    |           |
    |- 1 ---|
    |           |
    |- ... -|
    |           |
    |- 9 ---|
    |           |
    |- A ---|
    |           |
    |- B ---|
    |           |
    |- C ---|
    |           |
    |- D ---|
    |           |
    |- E ---|
    |           |
    |- F ---|
```

```
HOLLERITH CONSTANT

                    |<-----------------|
                    |                  |
--<width>-- H ---<EBCDIC character>----------------------------------->|
```

```
CHARACTER CONSTANT

          |<-----------------------|
          |                        |
---- ' ---/256\-<EBCDIC character>--- ' ------------------------------>|
  |                                      |
  |       |<-----------------------|     |
  |       |                        |     |
  |- " ---/256\-<EBCDIC character>--- " -|
```

RELATIONAL OPERATORS

```
<rel op>

---- .LT. ---------------------------------------------------------->|
   |          |
   |- .LE. -|
   |          |
   |- .EQ. -|
   |          |
   |- .NE. -|
   |          |
   |- .GT. -|
   |          |
   |- .GE. -|
```

ARITHMETIC OPERATORS

```
<arith op>

---- + ------------------------------------------------------------->|
   |        |
   |- - --|
   |        |
   |- * --|
   |        |
   |- / --|
   |        |
   |- ** -|
```

ARITHMETIC EXPRESSION

INTEGER EXPRESSION

REAL EXPRESSION

DOUBLE PRECISION EXPRESSION

```
            |<-<arith op>-|
            |             |
------------<operand>---------------------------------------------->|
   |       |
   |- + -|
   |       |
   |- - -|
```

```
<operand>

----<arithmetic constant>---------------------------------------------->|
  |                         |
  |-<constant name>-------------|
  |                         |
  |-<variable name>-------------|
  |                         |
  |-<array element name>--------|
  |                         |
  |-<function reference>--------|
  |                         |
  |- (<arithmetic expression>) -|
```

CHARACTER EXPRESSION

```
  |<------------ // ------------|
  |                             |
------<character constant>------------------------------------------->|
    |                        |
    |-<constant name>--------|
    |                        |
    |-<variable name>--------|
    |                        |
    |-<array element name>---|
    |                        |
    |-<substring name>-------|
    |                        |
    |-<function reference>---|
    |                        |
    |-<character expression>-|
```

RELATIONAL EXPRESSION

```
----<arithmetic expression>--<rel op>--<arithmetic expression>-------->|
  |                                                              |
  |-<character expression>--<rel op>--<character expression>---|
```

LOGICAL EXPRESSION

```
                    |<-------------------------------|
                    |                                |
--<logical operand>------------------------------------------------->|
                      |                             |
                      |- .AND. ---<logical operand>-|
                      |         |
                      |- .OR. --|
                      |         |
                      |- .NEQ. -|
                      |         |
                      |- .EQ. --|
```

<logical operand>

```
---------------<logical constant>------------------------------------->|
  |        |  |                        |
  |- .NOT. -| |-<constant name>----------|
              |                        |
              |-<variable name>----------|
              |                        |
              |-<array element name>-----|
              |                        |
              |-<function reference>-----|
              |                        |
              |-<relational expression>--|
              |                        |
              |- (<logical expression>) -|
```

DIMENSION Statement

```
              |<-------- , --------|
              |                    |
-- DIMENSION ---<array declarator>------------------------------------>|
```

<array declarator>

```
                    |<------------ , ------------|
                    |                            |
--<array name>-- ( ---/7\-<dimension declarator>--- ) ---------------->|
```

<dimension declarator>

```
-------------------------------<bound expression>--------------------->|
  |                           |
  |-<bound expression >-- : -|
```

<bound expression>

```
              |<---------<arith op>---------|
              |                             |
--------------<integer constant>--------------------------------------->|
  |- + -|   |-<integer constant name>-|
  |- - -|   |-<integer variable name>-|
            |- (<bound expression>) --|
```

EQUIVALENCE Statement

```
                  |<---------------------- , ----------------------|
                  |                                   |<------ , ------|    |
-- EQUIVALENCE --- ( --<equiv entity>-- , ---<equiv entity>--- ) ----->|
```

<equiv entity>

```
----<variable name>---------------------------------------------------->|
  |                              |
  |-<array name>---------|
  |-<array element name>-|
  |-<substring name>-----|
```

COMMON Statement

```
-- COMMON ------------------------------------------------------------->

  |<--------------------------------------------------------------|
  |                                |<---------- , ----------|      |
  |                                |                        |      |
>-----------------------------------<variable name>----------------->|
     |- / <common block name> / -|  |-<array name>-------|
                                    |-<array declarator>-|
```

Type Statement - Integer, Real, Double Precision, Complex, Logical

```
---- INTEGER ----------------------------------------------------------->
  |- REAL -------------|
  |- DOUBLE PRECISION -|
  |- COMPLEX ----------|
  |- LOGICAL ----------|

  |<--------------------- , ---------------------|
  |                                              |
>---<variable list>------------------------------------------------->|
                    |- / <initial value list> / -|
```

<variable list>

```
  |<---------- , ----------|
  |                        |
------<variable name>---------------------------------------------->|
    |-<array declarator>-|
    |-<array name>-------|
    |-<function name>----|
    |-<constant name>----|
```

Type Statement-Character

```
-- CHARACTER ------------------------------------------------------------>
              |- * --<length specification>-| |- , -|

  |<---------------------- , ----------------------|
  |                                                |
>---<variable list>------------------------------------------------------>|
                |- / <initial value list> / -|
```

<variable list>

```
  |<------------------------- , -------------------------|
  |                                                      |
------<variable name>-------------------------------------------------->|
    |-<array declarator>-| |- * --<length specification>-|
    |-<array name>-------|
    |-<function name>----|
    |-<constant name>----|
```

<length specification>

```
---- ( -- * -- ) ------------------------------------------------------>|
  |-<nonzero unsigned integer constant>-|
  |- (<integer constant expression>) ---|
```

IMPLICIT Statement

```
                |<--------------------- , ---------------------|
                |                                              |
                |             |<------------ , ------------|   |
                |             |                            |   |
-- IMPLICIT ---<type>-- ( ---<letter>--------------------- ) --------->|
                                        |- - --<letter>-|
```

```
<type>

---- INTEGER ------------------------------------------------------>|
   |                                         |
   |- REAL ----------------------------------|
   |                                         |
   |- DOUBLE PRECISION ----------------------|
   |                                         |
   |- COMPLEX -------------------------------|
   |                                         |
   |- LOGICAL -------------------------------|
   |                                         |
   |- CHARACTER -----------------------------|
                  |                          |
                  |- * --<length specification>-|
```

PARAMETER Statement

```
                    |<-------------------- , -------------------|
                    |                                           |
-- PARAMETER -- ( ---<constant name>-- = --<constant expression>--- ) >|
```

EXTERNAL Statement

```
                 |<--------------- , ---------------|
                 |                                  |
-- EXTERNAL -----<subroutine name>------------------------------------>|
                  |                                  |
                  |-<function name>------------------|
                  |                                  |
                  |-<block data subprogram name>-----|
```

INTRINSIC Statement

```
                    |<------- , ------|
                    |                 |
-- INTRINSIC ---<function name>--------------------------------------->|
```

SAVE Statement

```
          |<---------------- , --------------|
          |                                  |
-- SAVE ---------------------------------------------------------------->|
            |                               |
            |-<variable name>---------------|
            |                               |
            |-<array name>------------------|
            |                               |
            |- / --<common block name>-- / -|
```

DATA Statement

```
          |<---------------------- , ----------------------|
          |                                                |
-- DATA ---<variable list>-- / --<initial value list>-- / ------------>|
```

<variable list>

```
   |<------------- , -------------|
   |                              |
------<variable name>---------------------------------------------------->|
     |                          |
     |-<array element name>-----|
     |                          |
     |-<array name>-------------|
     |                          |
     |-<substring name>---------|
     |                          |
     |- <data implied-DO list> -|
```

<data implied-DO list>

```
-- ( ---<array element name>--- , <DO  variable> = <ae> , <ae> -------->
       |                      |
       |- <implied-DO list> --|

>------------ ) ---------------------------------------------------->|
  |          |
  |- ,<ae> - |
```

```
<ae>

--<integer constant expression>--------------------------------------->|


<initial value list>

  |<--------------------- , ---------------------|
  |                                              |
------------------------------<constant>----------------------------->|
    |                                   |
    |-<repeat>---------- * -| |-<constant name>-|
    |                       |
    |-<constant name>-|


<repeat>

--<nonzero unsigned integer constant>----------------------------------->|


Arithmetic Assignment Statement

----<variable name>-------- = --<arithmetic expression>--------------->|
  |                       |
  |-<array element name>-|


Logical Assignment Statement

----<variable name>-------- = --<logical expression>------------------>|
  |                       |
  |-<array element name>-|


Character Assignment Statement

----<variable name>-------- = --<character expression>---------------->|
  |                       |
  |-<array element name>-|
  |                       |
  |-<substring name>-----|
```

Statement Label Assignment Statement

```
-- ASSIGN --<label>-- TO --<variable name>---------------------------->|
```

Unconditional GO TO Statement

```
-- GO -- TO --<label>------------------------------------------------->|
```

Assigned GO TO Statement

```
-- GO -- TO --<variable name>----------------------------------------->|
                                  |                          |
                                  |         |<--- , --|      |
                                  |         |         |      |
                                  |------- ( ---<label>--- ) -|
                                  |     |
                                  |- , -|
```

Computed GO TO Statement

```
                      |<--- , --|
                      |         |
-- GO -- TO -- ( ---<label>--- ) ---------<integer expression>-------->|
                                  |     |
                                  |- , -|
```

Arithmetic IF Statement

```
-- IF -- ( --<arithmetic expression>-- ) --<label>-- , --<label>-- , -->

>-<label>------------------------------------------------------------->|
```

Logical IF Statement

```
-- IF -- ( --<logical expression>-- ) --<executable statement>-------->|
```

Block IF Statement

```
-- IF -- ( --<logical expression>-- ) -- THEN ----------------------->|
```

ELSE IF Statement

```
-- ELSE IF -- ( --<logical expression>-- ) -- THEN ------------------->|
```

ELSE Statement

```
-- ELSE --------------------------------------------------------------->|
```

END IF Statement

```
-- END IF ------------------------------------------------------------->|
```

DO Statement

```
-- DO --<label>-- <variable name> -- = --<arithmetic expression>-- , -->

>-<arithmetic expression>---------------------------------------------->|
                           |                               |
                           |- , --<arithmetic expression>- |
```

CONTINUE Statement

```
-- CONTINUE ----------------------------------------------------------->|
```

STOP Statement

```
-- STOP --------------------------------------------------------------->|
          |                      |
          |-<character constant>-|
          |                      |
          |  |<---------------|  |
          |  |                |  |
          |--- /5\ --<digit>-----|
```

PAUSE Statement

```
-- PAUSE ------------------------------------------------------------->|
          |                             |
          |-<character constant>-|
          |                             |
          |  |<---------------|         |
          |  |                |         |
          |--- /5\ --<digit>-----|
```

END Statement

```
-- END --------------------------------------------------------------->|
```

WRITE, READ, PRINT, PUNCH Statements

```
---- WRITE ---<unit identifier>----------------------------------------->|
   |         |                  |                                |
   |- READ --|                  |- ,<io list> --------------|
   |         |                                                   |
   |- PRINT -|                                                   |
   |         |                                                   |
   |- PUNCH -|                                                   |
   |                                                             |
   |- WRITE --- (<control information list>) ----------------|
   |         |                                 |                 |
   |- READ --|                                 |- ,<io list> -|
```

```
<control information list>

  |<-------------------------- , ---------------------------|
  |                                                         |
------/1*\---<unit identifier>---------------------------------------->|
     |     |                   | |                           |
     |     |-<combined expr>---| |- , <format identifier> -  |
     |     |                                                 |
     |     |- UNIT = --<unit identifier>-------------------  |
     |                                                       |
     |-/1\- FMT = --<format identifier>--------------------- |
     |                                                       |
     |-/1\- REC = --<integer expression>-------------------- |
     |                                                       |
     |-/1\- END = --<label>--------------------------------- |
     |                                                       |
     |-/1\- ERR = --<label>--------------------------------- |
     |                                                       |
     |-/1\- IOSTAT = ---<integer variable name>------------- |
                      |                                      |
                      |-<integer array element name>-------- |


<combined expr>

-- <unit identifier> ' <integer expression> -------------------------->|


<io list>

  |<----------- , -----------|
  |                          |
------<expression>---------------------------------------------------->|
    |                    |
    |-<array name>-------|
    |                    |
    |-<io implied DO list>-|


<IO implied do list>

-- ( -- <io list> , <DO variable> = <ae> , <ae> ------------- ) ------>|
                                                 |            |
                                                 |- ,<ae> -   |
```

```
<format identifier>

----<label>------------------------------------------------------------>|
   |                        |
   |-<variable name>--------|
   |                        |
   |-<array name>-----------|
   |                        |
   |-<character expression>-|
   |                        |
   |- * --------------------|


<unit identifier>

----<integer expression>----------------------------------------------->|
   |                      |
   |-<variable name>------|
   |                      |
   |-<array name>---------|
   |                      |
   |-<array element name>-|
   |                      |
   |-<substring name>-----|
   |                      |
   |- * ------------------|
```

OPEN Statement

```
                 |<----------------------- , -----------------------|
                 |                                                  |
-- OPEN -- ( -----/1*\--------------<unit indentifier>----------------->
                   |      |- UNIT = -|                            |
                   |-/1\-<file attribute specifier>---------------|
                   |-/1\- ERR = --<label>-------------------------|
                   |-/1\- FILE = --<character expression>---------|
                   |-/1\- STATUS = --<character expression>-------|
                   |-/1\- ACCESS = --<character expression>-------|
                   |-/1\- FORM = --<character expression>---------|
                   |-/1\- RECL = --<integer expression>-----------|
                   |-/1\- BLANK = --<character expression>--------|
                   |-/1\- IOSTAT = ---<integer variable name>-----|
                                     |-<integer array element name>-|
>- ) ------------------------------------------------------------->|
```

```
<file attribute specifer>
--<file attribute>-- = <attribute value> ---------------------------->|
```

CLOSE Statement

```
                 |<----------------------- , -----------------------|
                 |                                                  |
-- CLOSE -- ( -----/1*\--------------<unit identifier>----------------->
                  |     |         |                               |
                  |     |- UNIT = -|                               |
                  |                                               |
                  |-/1\- ERR = --<label>--------------------------|
                  |                                               |
                  |-/1\- STATUS = --<character expression>--------|
                  |                                               |
                  |-/1\- IOSTAT = ---<integer variable name>------|
                                    |                             |
                                    |-<integer array element name>-|

>- ) ---------------------------------------------------------------->|
```

INQUIRE Statement

```
-- INQUIRE -- ( --------------------------------------------------------->

  |<--------------------------- , ----------------------------|
  |                                                            |
>-----/1*\--------------<unit identifier>------------------------ ) -->|
   |         |- UNIT = -|                                      |
   |-/1\-<file attribute specifier>------------------------|
   |-/1\- ERR = --<label>----------------------------------|
   |-/1\- IOSTAT = ------<integer variable name>-----------|
   |-/1\- NUMBER = --| |-<integer array element name>------|
   |-/1\- RECL = ----|                                     |
   |-/1\- NEXTREC = -|                                     |
   |-/1\- EXIST = ------<logical variable name>------------|
   |-/1\- OPENED = -| |-<logical array element name>-------|
   |-/1\- NAMED = --|                                      |
   |-/1\- NAME = ------------<character variable name>-----|
   |-/1\- ACCESS = ------| |-<character array element name>-|
   |- /1\-SEQUENTIAL = --|
   |-/1\- DIRECT = ------|
   |-/1\- FORM = --------|
   |- /1\-FORMATTED = ---|
   |- /1\-UNFORMATTED = -|
   |-/1\- BLANK = -------|
```

```
<file attribute specifier>

--<file attribute>-- = ---<array element>---------------------------->|
                          |-<variable>------|
```

BACKSPACE, ENDFILE, REWIND Statement

```
---- BACKSPACE ------------------------------------------------------->
   |              |
   |- ENDFILE ---|
   |              |
   |- REWIND ----|

>---<unit identifier>----------------------------------------------->|
   |                                                                  |
   |      |<----------------------- , ----------------------|         |
   |      |                                                  |         |
   |- ( -----/1*\--------------<unit identifier>--------------- ) -|
           |          |           |                         |
           |          |- UNIT = -|                         |
           |                                                |
           |- /1\ ERR = --<label>--------------------------|
           |                                                |
           |- /1\ IOSTAT = ---<integer variable name>------|
                              |                             |
                              |-<integer array element name>-|
```

FILE Statement

```
-- FILE --<unit identifier>-- ( <file list> ) ----------------------->|
```

<file list>

```
   |<---------------- , ----------------|
   |                                    |
------<file attribute specifier>---------------------------------------->|
     |                                  |
     |- FILE = <character expression> ---|
     |                                  |
     |- STATUS = <character expression> -|
     |                                  |
     |- ACCESS = <character expression> -|
     |                                  |
     |- FORM = <character expression> ---|
     |                                  |
     |- RECL = <integer expression> -----|
     |                                  |
     |- BLANK = <character expression> --|
```

```
<file attribute specifier>

--<file attribute>-- = <attribute value> ----------------------------->|
```

FORMAT Statement

```
--<label>-- FORMAT --<format specification>-------------------------->|
```

```
<format specification>

-- ( ------------------------ ) ------------------------------------->|
      |                      |
      |-<format specifiers>-|
```

PROGRAM Statement

```
-- PROGRAM --<program name>------------------------------------------>|
```

Statement Function Statement

```
--<function name>-- ( ---------------------- ) = <expression> ----->|
                     |                      |
                     | |<------- , ------| |
                     | |                 | |
                     |---<variable name>---|
```

FUNCTION Statement

```
------------ FUNCTION --<funtion name>--<func arg list>-------------->|
  |        |
  |-<type>-|
```

```
<type>

-------------------------------------------------------------------------->|
  |                                                                   |
  |- INTEGER ---------------------------------|
  |                                                                   |
  |- REAL ------------------------------------|
  |                                                                   |
  |- DOUBLE PRECISION ------------------------|
  |                                                                   |
  |- COMPLEX ---------------------------------|
  |                                                                   |
  |- LOGICAL ---------------------------------|
  |                                                                   |
  |- CHARACTER -------------------------------|
                  |                                                   |
                  |- * --<length specification>-|
```

```
<func arg list>

-------------------------------------------------------------------------->|
  |                                              |
  |- ( ------------------------------ ) -|
        |                               |
        | |<---------- , ----------|    |
        | |                        |    |
        |-----<variable name>------|
              |                 |
              |-<array name>-----|
              |                 |
              |-<procedure name>-|
```

SUBROUTINE Statement

```
-- SUBROUTINE --<subroutine name>--<sub arg list>--------------------->|
```

```
<sub arg list>

--------------------------------------------------------------------->|
  |                                                  |
  |- ( ------------------------------ ) -|
           |                            |
           | |<--------- , ---------| |
           | |                       | |
           |-----<variable name>------|
               |                   |
               |-<array name>-----|
               |                   |
               |-<procedure name>-|
               |                   |
               |- * -------------|
```

```
Subroutine ENTRY Statement

-- ENTRY --<subroutine name>--<sub arg list>-------------------------->|
```

```
Function ENTRY Statement

-- ENTRY --<function name>--<func arg list>--------------------------->|
```

```
CALL Statement

-- CALL --<subroutine name>--<call arg list>-------------------------->|
```

```
<call arg list>

--------------------------------------------------------------------->|
  |                                                  |
  |- ( ------------------------------ ) -|
           |                            |
           | |<--------- , ---------| |
           | |                       | |
           |-----<expression>---------|
               |                   |
               |-<array name>-----|
               |                   |
               |-<procedure name>-|
               |                   |
               |- * <label> ------|
```

Function Reference

```
--<function name>-- ( ----------------------------- ) --------------->|
                     |                            |
                     | |<--------- , ---------| |
                     | |                      | |
                     |-----<expression>---------|
                         |                  |
                         |-<array name>-----|
                         |                  |
                         |-<procedure name>-|
```

RETURN Statement

```
-- RETURN ------------------------------------------------------------>|
          |                        |
          |-<integer expression>-|
```

BLOCK DATA Statement

```
-- BLOCK DATA -------------------------------------------------------->|
              |                               |
              |-<block data subprogram name>-|
```

APPENDIX B

COMPARISON OF B 6000/B 7000 FORTRAN 77 AND ANSI FORTRAN 77 STANDARD

Following are B 6000/B 7000 enhancements to the ANSI FORTRAN 77 Standard:

1. The TYPE statement may contain an initial value list to initially define entities.

2. The character set includes three additional characters: (&) ampersand, (%) percent sign, and (") quote character.

3. In the character type statement exponentiation is permitted in the expression defining the length of the character entity.

4. Hollerith and hexadecimal constants are implemented.

5. The combination of complex and double precision values with the operators +,-,*,/ is permitted.

6. The exponentiation of a complex value by a real value or a double precision value is permitted.

7. Within a DATA statement there is greater flexibility in the type of constants that may be used to initialize variables.

8. A quote edit descriptor has been implemented.

9. A Z edit descriptor has been implemented.

10. The G edit descriptor may be used with integer and logical data.

11. The use of X is equivalent to 1X for the X edit descriptor.

12. A PUNCH statement has been implemented.

13. An alternate form is permitted for the combination of the unit and record number specifiers.

14. A unitless form of the WRITE statement has been implemented.

15. The digits 2 through 9 have been added to the standard carriage control characters. These indicate to skip to the same numbered format channel.

16. An alternate form of the CLOSE statement has been implemented.

17. A FILE statement is implemented to assign file attributes.

18. Additional intrinsic functions have been implemented to access time and data information, to perform boolean operations, and for pseudo-random number generation.

19. A FIND statement is implemented to permit positioning of a direct access file.

20. H edit descriptor may be used on input.

21. Subprograms may be recursive.

In addition to the above enhancements there are several differences in implementation of blank and named common between the B 6000/B 7000 FORTRAN 77 and the ANSI FORTRAN 77 Standard.

Entities in blank and named commons may be initially defined in a BLOCK DATA subprogram or in a main program. The ANSI implementation prohibited entities in blank common from being initially defined in a BLOCK DATA subprogram. It also prohibited entities in named common from being initialized in a main program.

With regard to I/O the following points should be noted:

1. All files are required to have names and default names are assigned to files, depending on the unit to which the file is connected.

2. Units 5, 6, and 7 exist whether or not they are used explicitly.

The ANSI FORTRAN 77 Standard does not specify a limit for the length of a character constant. A limit of 255 characters exists for a character constant in the B 6000/B 7000 implementation.

INDEX

Subject Page

A

B

C

Subject | Page

INDEX (CONT.)

Subject Page

E

INDEX (CONT.)

Subject | Page

F

G

H

I

INDEX (CONT.)

Subject | Page

INDEX (CONT.)

Subject Page

INDEX (CONT.)

Subject | Page

Documentation Evaluation Form

Title: B 6000/B 7000 Systems FORTRAN 77 Reference Manual

Form No: 5011729

Date: Oct. 1981

Burroughs Corporation is interested in receiving your comments and suggestions regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error

Comments:

From:

Name ______

Title ______

Company ______

Address ______

Phone Number ______ Date ______

Remove form and mail to:

Documentation Dept, TIO - East
Burroughs Corporation
Box CB7
Malvern, PA 19355