HP 3000

FORTRAN

HEWLETT hp PACKARD

3000 computer systems

HP 3000

FORTRAN

03000-90007 November 1972

List of Effective Pages

Pages	Effective Date
Title	Nov. 1972
Copyright	Nov. 1972
iii	Nov. 1972
v to ix	Nov. 1972
1-1 to 1-4	Nov. 1972
2-1 to 2-17	Nov. 1972
3-1 to 3-10	Nov. 1972
4-1 to 4-23	Nov. 1972
5-1 to 5-16	Nov. 1972
6-1 to 6-8	Nov. 1972
7-1 to 7-7	Nov. 1972
8-1 to 8-11	Nov. 1972
9-1 to 9-49	Nov. 1972
10-1 to 10-2	Nov. 1972
11-1 to 11-8	Nov. 1972
A-1 to A-2	Nov. 1972
B-1 to B-4	Nov. 1972
C-1	Nov. 1972
D-1 to D-19	Nov. 1972

Printing History

Part No.	Date	Update Package	Date
03000-90007	Nov. 1972		

PREFACE

The publication is the reference manual for the HP 3000 FORTRAN programming language (FORTRAN/3000). It describes the syntax and functions of each FORTRAN/3000 instruction.

Included in the manual are descriptions of the use of the FORTRAN file facility, FORTRAN Formatter, and the use of the non-FORTRAN (SPL/3000) language subprograms and system intrinsics. For more detailed information on the above subjects, refer to

- *HP 3000 Multiprogramming Executive Operating System (03000-90005)*
- *HP 3000 Compiler Library (03000-90009)*
- *HP 3000 Systems Programming Language (03000-90002)*

CONTENTS

TABLES

FIGURES

SECTION I

Introduction

The FORTRAN/3000 language is based upon the ANSI Standard FORTRAN (X3.9-1966). In addition, FORTRAN/3000 has many extensions which expand the capabilities and increase the power of the language as a problem-solving tool.

FORTRAN/3000 also incorporates many of the extensions implemented for previous versions of the Hewlett-Packard FORTRAN to maintain upward compatibility in FORTRAN programs written for the 2100 family of HP computers.

FORTRAN/3000 operates under the control of the HP 3000 Multiprogramming Executive operating system (MPE/3000). FORTRAN/3000 programs can be compiled using the minimum MPE/3000 hardware configuration.

PROGRAM ELEMENTS

Character Set

A FORTRAN/3000 source program is written using alphabetic characters A through Z, numeric characters 0 through 9, and the following special characters:

	Blank
=	Equal sign
+	Plus sign
-	Minus sign
*	Asterisk
/	Slash
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Decimal point
$	Dollar sign
" "	Quotation marks
'	Apostrophe
[	Left bracket
]	Right bracket
#	Hatch mark
%	Per Cent
@	Record pointer
:	Colon
\	Back slash
	"Any other printing ASCII character"

Blanks are used anywhere within a FORTRAN statement. They are ignored by the compiler except in Hollerith and string constants, where they represent blank characters. The significance of blanks outside the body of a FORTRAN statement is explained under "Statements," below.

Lines

Lines consist of a sequence of as many as 80 characters numbered from 1 to 80. Characters 73 through 80 are not part of the program text, but are used by some compiler options for sequencing information.

Statements

From one to twenty 72-character lines can be combined to form a statement. The first line of the statement contains a zero or blank in character position 6. A continuation line (any line of the statement following the first line) contains any character other than blank or zero in position 6. A continuation line cannot contain a C or $ in position 1 (see "Comment Lines" and "Control Records" below).

There are two main statement types—executable andd nonexecutable. Executable statements are assignment statements, control statement, and input/output statements. Nonexecutable statements are specification statements such as Type, COMMON, and IMPLICIT.

Statement Labels

A statement can be labeled so that it can be referred to by other statements in the program. A label consists of one to five numeric digits placed in any of the first five positions of a line. The number is unsigned and is in the range of 1 through 99999. Embedded spaces and leading zeros are ignored. If no label is used, the first five character positions of the line must remain blank.

Comment Lines

Lines containing comments can be included between statements. The comments are printed as part of the source program listing. A comment line requires a C in position 1 and treats positions 2 through 72 of the line as text. If more than one comment line is used, each line must contain a C in position 1. Comment lines cannot be inserted between lines of a single statement or single control record.

Control Records

Control records are not part of the program proper, but are included with the source program to indicate compiler options, such as suppressing listings or diagnostic messages. Control records are detailed in Section XI.

PROGRAM FORMAT

Fixed Field Format

A program unit entered in fixed field format consists of a set of 80-character lines; characters in a line are numbered from 1 to 80. Character positions 73 through 80, while not affecting the program logic, can be used for sequencing information. The order of these lines is the order in which they are entered into the input device.

The program is made up of comment lines, control record lines and ~~the~~ initial and continuation lines which make up the executable and nonexecutable statements of the program. Comments are indicated by the letter C in position 1. A control record indicates which options the compiler is to take when compiling the source program. A control record starts with a $ in position 1. Positions 2 through 72 are available for control options. Commas plus optional blanks separate each individual option. If a control record takes more than one line, the initial line and any succeeding lines which must be continued must end with an &. Each continuation line must begin with a dollar sign ($) in position 1.

The final line of any FORTRAN program unit must be an end line (see Section VI). An example of fixed field format is

CΔTHISΔISΔANΔEXAMPLEΔOF ΔFIXEDΔFIELDΔFORMAT	*Comment line*
$CONTROLΔNOLIST,ΔNOWARN	*Control record*
ΔΔΔΔΔΔ|INTEGER A(10),I	*FORTRAN Statement*
ΔΔΔΔΔΔ|DO 20 I=1,10	*FORTRAN statement*
020 ΔΔΔA(I)=I	*Statement with label in positions 1 to 5*
ΔΔΔΔΔΔ|END	*END line*

Free-Field Format

A program unit entered in free-field format consists of a set of 9 to 80-character lines. Each line of a free-field program begins with a sequence field (corresponding to positions 73 and 80 in fixed-field format). The sequence field extends up to (but not including the first blank in the line. A sequence string less than 8 characters long is treated as being right-justified by the compiler upon input with leading zeros set into the sequence field for unspecified characters. A blank in position 1 implies that the entire sequence field is blank.

The remaining positions in the line (up to 71 characters total), starting with the first position after the blank terminating the sequence string) make up the FORTRAN line. Comment lines are indicated by a hatch mark (#) in the first position following the sequence field. Control records are indicated by $ as their first character. If a control record or FORTRAN statement takes more than one line, the initial line and each additional line except the last of that record must be terminated with an &. Each control record continuation line begins with a $ following the sequence field and blank.

Statement labels may be used, but they need not start in the first position following the sequence string. If the first nonblank character following the sequence string is a digit, then it begins a statement label. The label can be more than five digits long, but cannot execeed 99999 in value. The first nonblank, nondigit character following the sequence field starts the body of the FORTRAN statement. For example

001Δ#ΔTHISΔISΔANΔEXAMPLEΔ OFΔFREEΔFIELDΔFORMAT	*Comment line; a blank separates the sequence field from the rest of the line.*
002Δ$CONTROLΔNOLIST,ΔNOWARN	*Control record.*
00000003ΔINTEGER A(10),I	*No space need be left for labels if the line does not contain one.*
Δ4DO 20 I = 1,10 05 20 A(I) = I	*Label follows the first blank after sequence.*
0006 END	*END line.*

SECTION II

Data Elements

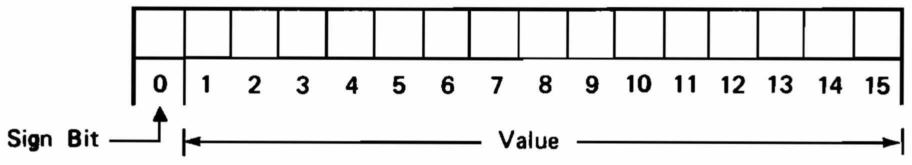
DATA STORAGE FORMATS

FORTRAN/3000 processes six types of data—integer, logical, real, double precision, complex, and character. Each data type differs in the way it is internally represented in memory.

Integer Format

Integer values are stored in one 16-bit computer word. The leftmost bit represents the arithmetic sign of the number (1 = negative, 0 = positive). The other 15 bits represent the binary value of the number. Numbers are represented in two's complement form.

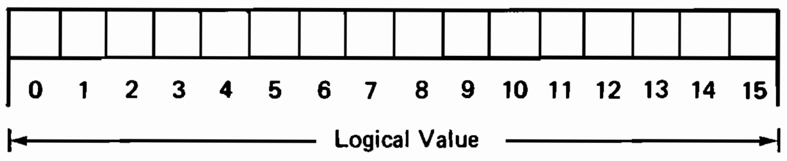
Figure 2-1. Internal Representation of Integer Values



Logical Format

Logical values are stored in one 16-bit computer word. Bit 15 is used to determine the TRUE/FALSE (Boolean) value.

Figure 2-2. Internal Representation of Logical Values



Real (Floating Point) Format

Real numbers are stored in two consecutive 16-bit computer words as normalized fractions with a scale factor and a sign bit (which applies to the entire number). The fraction (F) is always positive. The sign bit (S) contains the sign of the number (S = 0 for positive, S = 1 for negative). The binary point is to the left of bit 10 with an implied leading 1 to the left of the binary point. E represents (scale factor + 256_{10}). The formula for the decimal value of a floating point representation is

$$\text{Decimal value} = (-1)^{S} \times 2^{(E-256)} \times (1 + F \times 2^{(-22)})$$

Figure 2-3. Internal Representation of Real Values

S	E								F						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Word 1

F															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Word 2

Represents Zero (0) in Memory

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Word 1

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Word 2

Represents One (1) in Memory

0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Word 1

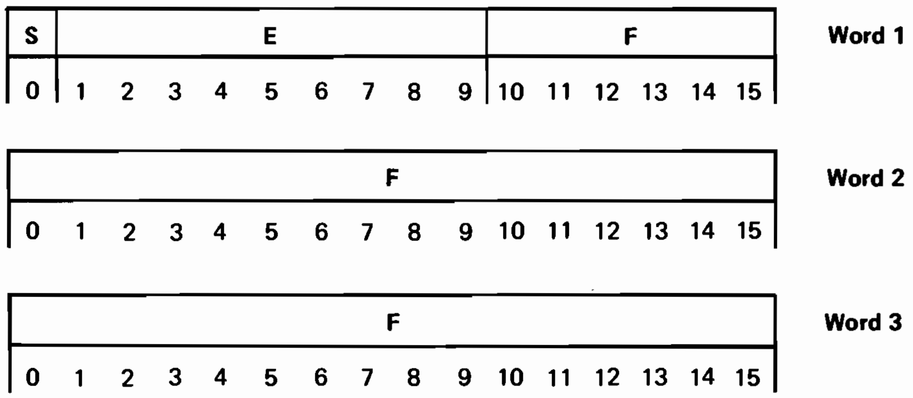
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Word 2

Double Precision Real Format

Double precision values are stored in three consecutive 16-bit computer words and are similar to real (single precision) values, except that the fractional part of the number is extended from 22 to 38 binary bits.

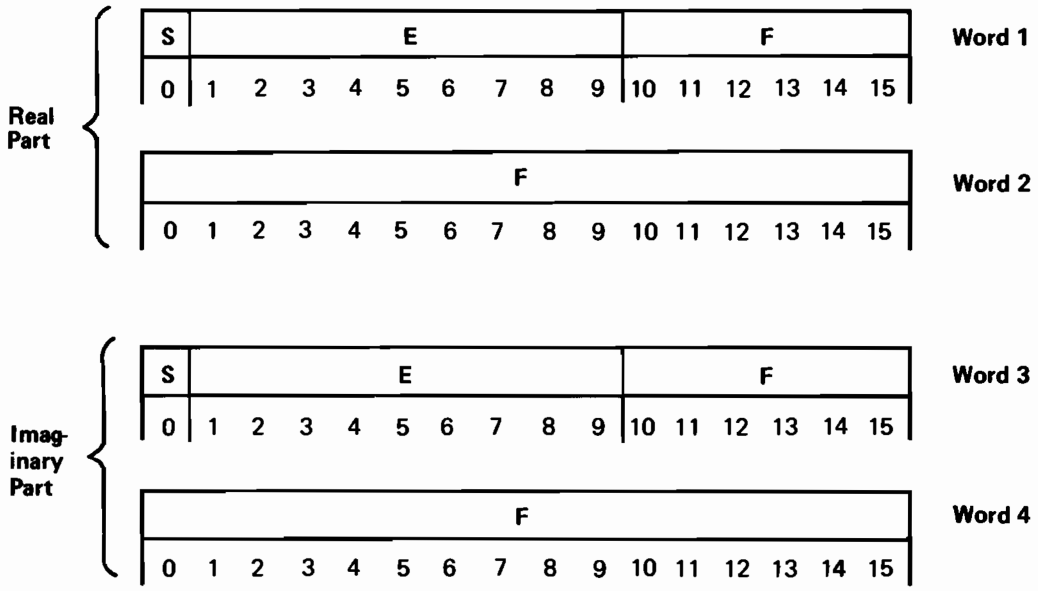
Figure 2-4. Internal Representation of Double Precision Real Values



Complex Format

A complex value consists of an ordered pair of real numbers, specifying the real and imaginary parts. Complex numbers are stored in four consecutive 16-bit computer words; the first two words for the real part, the second two words for the real number specifying the imaginary part. The two numbers specifying the complex value must be real

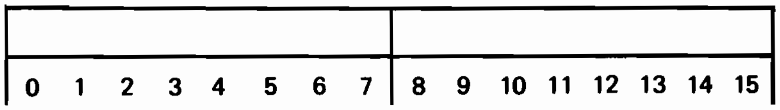
Figure 2-5. Internal Representation of Complex Values



Character Format

Character values are represented by 8-bit ASCII codes, two characters packed in one 16-bit computer word. The number of words used to represent a character value depends upon the actual number of characters in the source representation of the value.

Figure 2-6. Internal Representation of Character Values



CONSTANTS

A constant is a data element representing one specific value which remains unchanged throughout the program. A constant's type and specific value are determined by its representation in the source program.

Integer Constants

Integers are whole numbers Containing no fractional part. Integers may be specified in four ways: decimal, octal, ASCII, and composite.

Decimal integer constants use the decimal digits 0 through 9. They can contain a leading plus (+) or minus (–) sign (a number with no leading sign is positive). The number ranges from -32768 to 32767. For example,

```
+45
-365
4012
```

Octal integer constants are denoted by the % character. They may contain up to six octal digits and an optional leading sign. The number ranges from -100000_8 to $+77777_8$. For example,

```
 #4777
+%605
-%17
%177777
```

ASCII integer constants are used to write one or two ASCII character bit patterns into one 16-bit computer word. ASCII integer constants are written with the % character followed by the ASCII characters enclosed in quote marks (also called string bracket characters) for apostrophes. For example,

```
+%"AB"
-%"U"
%"HI"
%"L"
-%'XY'
+%'A'
```

If only one ASCII character is specified, the bit pattern representing that character is placed in the computer word right-justified, and the left half of the word is filled with leading zeros. Leading plus or minus signs are allowed.

Integer constants can also be specified by composite numbers. See "Composite numbers," this section.

Real (Floating Point) Constants

Real constants are represented by an integer part, a decimal point and a decimal fraction part (with an optional leading sign). The constant can contain a scale factor (which represents a power of ten by which the constant is multiplied). The eleven forms of a real constant are

n.	*n*E±*e*
.*n*	*n*E*e*
n.*n*	.*n*E*e*
n.*E*+*e*	*n*.E*e*
.*nE*+*e*	*n*.*n*E*e*
n.*n*E±*e*	

n is a decimal integer. The construct E±*e* stands for $10^{\pm e}$, which is multipled by the other part of the number (*n*., .*n*, *n*.*n*, etc). The construct E*e* is equivalent to E±*e*.

3.4E-4 = 3.4×10^{-4} = .00034

-3.4E4 = -3.4×10^{4} = -34000

A real constant may be written any number of digits in length, but the internal representation in memory only allows six or seven significant decimal digits.

Real constants also can be represented by octal numbers, followed by the letter R. The bit pattern specified by the octal number is loaded (right-justified) into two consecutive 16-bit words in memory and is treated as a floating point number.

%3775R = $1.00017764_8 * 2^{-256}$

ASCII real (right-justified) constants also are allowed. From one to four 8-bit ASCII patterns are stored in the two 16-bit words.

```
%"ABCD"R
-%"DEF"R
%"V"R
+%"XZ"R
```

Composite numbers followed by the letter R also can specify real numbers. See "Composite Numbers" in this section.

Double Precision Real Constants

Double precision real constants are similar to real (single precision) constants. Substituting the letter D for the letter E in the scale factor of a real constant gives a double precision real constant with 10 or 11 significant decimal digits as opposed to the 6 or 7 significant digits in the single precision real constant. (Double Precision Constants start with an optional sign.)

The eight decimal representations of double precision constants are

*n*D±*e*	*n*D*e*
.*n*D±*e*	*n*.D*e*
n.D±*e*	*n*.D*e*
n.*n*D±*e*	*n*.*n*D*e*

The real constant forms .*n*, *n*., or *n*.*n* (those without the scale factor) are not allowed for double precision constants, as FORTRAN has no way of knowing whether the number should be stored in single or double precision format.

When written, double precision octal numbers are preceded by % and are followed by the letter D. For example,

```
-%3776125D
%64333D
%45D
```

ASCII double precision real constants also are allowed. From one to six 8-bit ASCII patterns are stored in the three 16-bit words.

```
%"ABCDEF"D
-%"A"D
%"TR"D
+%"DFG"D
%"LKJH"D
```

Composite double precision real constants are also allowed. See "Composite Numbers" in this section.

Complex Constants

Complex constants are represented by an ordered pair of real constants enclosed in parentheses and separated by a comma. The first number represents the real part and the second number represents the imaginary part of the complex number.

The real constants of each ordered pair can be represented as integers, decimal fractions (with or without a scale factor), octal numbers, or composite numbers.

Double precision constants cannot be used to represent either the real or the imaginary part of a complex constant. For example,

```
(3.0, -2.5E3)

(%376R, %736R)
```

Logical Constants

Only two values are normally used for logical constants: .TRUE. and .FALSE. However, the internal representation of these logical values allows use of a full 16-bit word. .TRUE. is represented by all 16 bits equal to 1. .FALSE. is represented by all 16 bits equal to 0. Any other pattern of 16 bits, can be used with logical operators to perform masking operations (see Section III).

The actual bit pattern of a mask is specified by an octal constant, an ASCII character string of up to two ASCII characters or a composite number followed by an L.
For example,

```
%177777L
%“AB”L
%1006L
```

Character Constants

Character constants represent ASCII character strings which can be manipulated using character expressions and input/output statements. String constants are character values bound by quote marks or apostrophes, called string bracket characters. Blanks are significant characters within a character string. For example,

```
“THIS IS A STRING”
“NOW IS THE TIME”
'ANOTHER FORM'
'HE SAID, “HELLO”'
```

If a quote mark (“) must be included within a string bracketed by quotes, or if an apostrophe (′) must be included within a string bracketed by apostrophes, write it twice in a row to distinguish it from a string bracket. Apostrophes in strings bracketed by quotes and quotes in strings bracketed by apostrophes need only be written once. For example,

```
“ABC”“XYZ”          'AB″CD'          “AB'CD”
“4XC”“D”            'AB″CD'
```

To indicate a null string (a string with no value) write a pair of string bracket characters with no intervening characters (e.g., “”).

NOTE: *A character string written with one or more blanks between the string delimiters is not the null string, but represents a string of ASCII blanks.*

Hollerith constants consist of a decimal integer specifying the length of the constant, followed by the letter H and the character value. For example,

Hollerith		ASCII		String
19HBLANKS ARE INCLUDED	=	BLANKS ARE INCLUDED	=	“BLANKS ARE INCLUDED”
7HAB“CDFG	=	AB“CDFG	=	“AB”“CDFG”

A single character can be specified by an octal number representing a character bit pattern. The octal number is followed by the letter C:

```
%101C
%16C
%207C
```

This form is useful for representing nonprinting characters (such as carriage returns) in source programs.

Composite Numbers

Composite numbers are a convenient way of representing specific bit patterns for any type constant except character or complex. A composite number takes the form

% [*bit pattern*] *letter*

where % is an octal number indicator and *bit pattern* consists of one or more subfields in the form

$A_1/N_2, A_2/N_2, ..., A_n/N_n$

A_1 through A_n are decimal integers which represent the number of bits in the bit pattern subfield. N_1 through N_n are the octal or decimal values set right-justified into the subfields. Unspecified leading bits are set to zero. Extra leading bits are truncated.

For example, 3/7 creates a subfield 3 bits long with the binary value 111_2 set into it. 4/7 creates a subfield 4 bits long. The value $7_{10} = 111_2$ is loaded right-justified into the four-bit field and the unspecified leading bit is set to zero. The resulting subfield is 0111_2.

letter indicates the data storage word format for the type of constant. Integer composite constants do not have a letter suffix. Logical is indicated by L, real by R, double precision real by D. Complex constants consist of an ordered pair of real numbers, either or both of which can be real composite numbers. The bit pattern specified in each subfield is concatenated from left to right and the result is stored right-justified in the storage space for the constant type indicated by the *letter*. Unspecified leading bits are set to zero.

Type	Example	
Integer	%[4/15,6/%13,2/1]	*(no letter)*
Logical	%[12/64]L	
Real	%[4/9,16/%1245,10/49]R	
Double precision real	%[35/4775,10/%777]D	
Complex	(%[15/777]R,%[12/%4444]R)	

For example,

%[3/7,4/7]L

where L indicates a logical constant.

The two subfields $3/7 = 111_2$ and $4/7 = 0111_2$ are concatenated left to right to form the bit pattern 1110111_2.

1110111_2

This value is placed right-justified in a 16-bit word, with unspecified leading bits set to zero. The resulting logical value is

0000000001110111_2

Another example is

%[3/7,4/7]R

where R indicates a real (floating point) constant. The bit pattern specified by the two subfield 3/7 and 4/7

1110111_2

is placed right-justified into a 32-bit word, with unspecified leading bits set to zero. The resulting real value is

$00000000000000000000000001110111_2$

VARIABLES

A variable is a symbolic name from one to 15 alphameric characters (the first character must be a letter) capable of representing any value of the type associated with the variable name.

The type of a variable can be determined by a Type statement (see Section IV). If the variable is not mentioned in a Type statement, the variable type is determined by the first letter of the variable name. Names starting with I, J, K, L, M, or N are type integer. Variable names starting with any other letter are type real.

The value of a variable is given through an assignment statement or a READ statement. The specific value represented by the variable may be changed during execution of the program containing the variable.

Simple Variables

A simple variable is a symbolic name which has one, and only one, value at one time. For example,

INTEGER	REAL
I	ALPHA
JAM1234	G99887766
NOWHERE4567	ZERO
L1	Q45

Arrays

An array is a collection of one or more values of the same type, all represented by the same symbolic name. An array variable is an array name suffixed by a subscript to designate exactly one value of the collection. An array variable is therefore equivalent in usage to a simple variable.

Array type can be determined by a Type statement (see Section IV). If the array name is not mentioned in a Type statement, the array type is determined by the first letter of the variable name. Names starting with I, J, K, L, M, or N are type integer. Array names starting with any other letter are type real. An array and its dimensions must be declared in a DIMENSION, Type, or COMMON statement (see Section IV).

SUBSCRIPTS

Subscripts point to a specific element of a named array. A maximum of 255 array dimensions are allowed; an array variable must contain as many subscript expressions as the array's definition specifies. Subscripts are written in the form:

$(exp_1, exp_2, ..., exp_n)$

where exp_i is an arithmetic constant, variable, or expression of any type except complex. The computed value of any subscript is truncated to the nearest integer, regardless of the original implicit or explicit type. Examples of array variables are:

INTEGER	**REAL**
I(J,K)	Q22(4,5,6,7,8)
LAD12(3,4,5,7)	YOU(4+LT,3*X+4,6,5)
JO(4*I,6+HA,7)	RUN(6)
NO(3,5/1+K,8*T)	ALL(L*K*6,7-24*FUN)

FUNCTION REFERENCES

A function reference is a symbolic name from one to 15 alphmeric characters (the first of which must be alphabetic) followed by a list of arguments. The symbolic name references a FORTRAN computational process (defined elsewhere) which is designed to return a value assigned to the symbolic name. A function reference takes the form

name (param, param, ..., param)

name	the symbolic name of the statement function, intrinsic function or function subprogram.
param	a variable name, array name, array element, function subprogram name, subroutine subprogram name, Hollerith constant, expression, or an arithmetic or logical expression bounded by back slashes (\ ).

The actual subset of arguments allowed for each type of function (statement function, intrinsic function, or function subprogram) is described in "Dummy and Actual Argument Characteristics" in this section.

A function reference requests a specific value of the type assigned to it and is equivalent in usage to a variable reference of the same type. When a function reference is encountered during the evaluation of an expression, control is passed to the function. The function is executed using actual arguments listed in the function reference in place of the dummy arguments in the function definition. The function name is assigned a value depending upon the values of the actual arguments. (For discussions of dummy arguments and function definition, see "Dummy and Actual Argument Characteristics" in this section and "Function Subprograms" in Section VI.) After the function is executed, control passes back to the expression in the calling program. Evaluation of the expression containing the function reference then continues, as shown in the following example.

```
PROGRAM MAINPROG                 INTEGER FUNCTION ICOUNT(IA,IB)
MAX = 6                          ICOUNT = (IA + 6)/IB
LIMIT = 6                        END
.
.
.
Z = 42 + ICOUNT (MAX,LIMIT)
.
.
.
END
```

In this example, the function ICOUNT is assigned the value 2 and Z is assigned the value 44. The values of MAX and LIMIT replace the dummy arguments in ICOUNT's definition. ICOUNT is assigned a value and this value is used to evaluate the expression.

The type associated with a function name is determined by a Type statement (Section IV). If the function name does not appear in a Type statement, the function type is determined by the first letter of the function name. Function names beginning with I, J, K, L, M, or N are type integer, while names beginning with any other letter are type real. (This typing convention can be modified by an IMPLICIT statement.

A function reference invokes either a local or global function. A local function is recognized only within the program unit which defines it. A globally defined function is recognized in any program unit which does not define its name for some local purpose. Statement functions (Section IV) are local functions; their names can never appear in an EXTERNAL statement nor be referenced outside the defining program unit. Intrinsic functions (Section VII) are local functions whose names are predefined to the compiler; they are implicitly defined within each program unit which does not redefine their names in a specification statement or through implicit use as a simple variable. Function subprograms including basic external functions (Section VI) are global functions. They may appear in any program unit which does not redefine the name locally.

Individual characteristics of local and global function references appear below.

STATEMENT FUNCTION REFERENCE

A statement function reference invokes a computational process local to the calling program unit (Section IV). The actual arguments allowed n a statement function reference are constants, simple variables, array elements, function references or expressions consisting of any combination of the above. Actual arguments of type character are not allowed. For example,

```
ICOUNT(A + 6,XRAY)

PUSHSTAK(6,A(3),ICOUNT(X,Y,Z))

FLIP(MAX,DIX,DIL)
```

FUNCTION SUBPROGRAM REFERENCE

A function subprogram reference invokes a user externally defined computational process or a basic external function (Section VI). A function reference is similar in usage to a simple variable reference and can be used in any expression where a constant, variable or (expression) can be used. Function subprograms are global functions.

Outside of a reference the function name can only appear in a Type or EXTERNAL statement or as an actual parameter for a subroutine CALL statement or another function reference.

If the function name is used as an actual argument in a function reference, the function name must be identified either by including the name in an EXTERNAL statement or by suffixing empty parentheses "( )" after the function name. A function name cannot appear in an EXTERNAL statement within the program unit which defines it (i.e., a program unit headed by a FUNCTION statement using the function name). If such a function name appears as an actual argument within the program unit, providing its definition and is not suffixed by "( )", the value of the function will be passed as a simple variable of its type. For example,

```
   PROGRAM EX
   DIMENSION TOTAL (10)
   REAL TOTAL
   EXTERNAL ADDUP
   DO 40 I = 1,10
40 TOTAL(I) = (ADDUP(I,I+1) +6)/4
   END

   REAL FUNCTION ADDUP(J,K)
   INTEGER J,K
   ADDUP = J + K
   RETURN
   END
```

In the main program (EX), statement 40 references a real function named ADDUP. Each time the expression on the right-hand side of the assignment statement (statement 40) is evaluated, control passes to ADDUP. The actual arguments of the function at the time of the reference are I and I + 1. The simple variables used as dummy arguments in the FUNCTION statement (J and K in this example), use expressions of the same type as actual parameters (I and I + 1 are both integer expressions). When control passes to the function, the values of I and I + 1 are used in place of the dummy parameters J and K. The value is given to the function name through an assignment statement which uses the function name as a simple variable in the left side. Each time the range of the DO loop in the main program is executed, the value of ADDUP changes since the actual arguments change each time the function is referenced.

When functions are used, care should be taken to avoid side effects of parameter modification during execution. For example,

```
   PROGRAM SAM
   REAL DIX
   EXTERNAL HUF
   R = 4
   DIX = 7
   ALF = DIX + HUF (R,DIX) + R
   END

   REAL FUNCTION HUF(S,T)
   A = S * 6
   T = A * 4
   HUF = T + 1
   END
```

In the preceding example, function HUF redefines one of its actual arguments during the course of its execution. The function reference HUF(R,DIX) (R=4, DIX = 7) results in the following calculations:

A = R * 6 = 4 * 6 = 24

DIX = A * 4 = 24 * 4 = 96

HUF = DIX + 1 = 96 + 1 = 97

Order of evaluation of the expression DIX + HUF(R,DIX) + R in the main program is not necessarily left to right; the results are not necessarily predictable.

INTRINSIC FUNCTIONS

Intrinsic functions are computational procedures which perform useful operations, such as converting integers to real values. The names of intrinsics are predefined to the FORTRAN/3000 compiler. To use an intrinsic, it is necessary to reference the intrinsic function name (along with appropriate arguments) in an expression. It is incorrect to mention the intrinsic function name in an EXTERNAL statement unless the program intends to redefine its name as a procedure subprogram.

An intrinsic function name can be redefined for other use within a program unit in several ways. One way is to use the name in a Type statement different from the function name's normal type. Another way is to use the name in a DIMENSION, COMMON, EXTERNAL, EQUIVALENCE, SUBROUTINE, or FUNCTION statement. Then the intrinsic name can be used as a simple variable or array name, statement function name, or subprogram name. This redefinition applies only to the program unit containing the redefining specification statement. The original FORTRAN compiler-defined intrinsic is used in any other program unit that does not redefine the name. An intrinsic name can be redefined in a SUBROUTINE or FUNCTION statement. The newly defined subroutine or function then can be used in the defining program unit and in any program unit using the subprogram name in an EXTERNAL statement. Any function reference using an intrinsic name will call the FORTRAN-defined intrinsic unless the name is used in an EXTERNAL statement or a Type statement of a differing type.

Intrinsics are considered locally defined functions. A list of all the intrinsics their uses and arguments, appears in Section VII. For more detail on intrinsics, consult *HP 3000 Compiler Library (03000-90009).*

DUMMY AND ACTUAL ARGUMENT CHARACTERISTICS

Actual arguments in a subroutine call or function reference must agree in number, order, and type with the dummy arguments they replace.

Within subprograms, dummy arguments may consist of simple variables, array names, subroutine names, or function names, Dummy arguments are local to the subprogram or statement function containing them; they can be the same as names appearing elsewhere in the program.

No element of a dummy argument list can occur in a COMMON, EQUIVALENCE, or DATA statement. When an array name is used as a dummy argument, the dummy array name must be mentioned in a DIMENSION statement within the body of the subprogram.

In subroutine subprograms, a list of asterisks (*) may follow the list of dummy arguments. When the subprogram is called using actual arguments, statement labels (prefixed by $'s) are substituted for the asterisks to indicate optional return points in the calling program. The mechanism for choosing optional return points is described in "Return Statement", Section V.

Actual arguments appearing in subroutine calls or function references may be

- A constant
- A variable name
- An array name
- An array element
- A subroutine name
- A function subprogram name
- An expression

A dummy simple variable requires an actual argument consisting of an expression of the simple variable's type. If the expression consists solely of a simple variable, the variable name is transferred to the subprogram. The variable can be redefined by the subprogram. If the expression is more complicated (not a simple variable), the subprogram should not modify the dummy argument associated with the expression.

A dummy array argument requires an actual argument consisting of an array name or an array variable of the dummy array's type. The number of subscripts (dimensions) of the actual array need not match the number in the dummy array. The elements of the dummy and actual array are dynamically equivalenced using the array successor function (see Section IV).

A dummy subroutine or function subprogram name requires an actual argument consisting of an external function subprogram of the same type or a subroutine subprogram name. The function or subroutine name must be declared in an EXTERNAL statement within the body of the calling program unit unless the name is a recursive call within the defining program unit. If not so declared, the function or subroutine name in the actual argument list must be suffixed with empty parentheses "( )". A statement function name or an intrinsic function cannot be used as an actual argument, although an expression used as an actual argument can contain reference to them as part of the expression.

Dummy simple variables of type character may use Hollerith constants or string variables as actual arguments.

Note: *FORTRAN subprograms accept all arguments by reference to the resulting value of the argument. No FORTRAN-written subprogram receives actual arguments by value. In order to facilitate invoking non-FORTRAN language subprograms which do allow passing of arguments by value, expressions enclosed in a pair of "\" can be used as actual arguments. This construct tells FORTRAN to pass the argument by value.*

NUMBER RANGES

Numbers represented by FORTRAN constants and variables have specific positive and negative ranges which limit the size of the number represented. Table 2-1 shows the various number types and their associated ranges. Complex numbers are not shown; they are represented by an ordered pair of real numbers.

Table 2-1. Number Ranges in FORTRAN

Type	Number of bit in memory	Range	Comments
Integer	16	$-32{,}768_{10}$ to $+32{,}767_{10}$	Negative integers represented in two's com-complement form.
Logical	16	0_{10} to $65{,}535_{10}$	Treated as bit patterns
Real (Floating Point)	32	$-2^{256} \times (2-2^{(-22)})$ to $+2^{256} \times (2-2^{(-22)})$	Six to seven decimal digits accuracy with a a range of -10^{77} to $+10^{77}$.
Double Precision Real	48	$-2^{255} \times (2-2^{(-38)})$ to $+2^{255} \times (2-2^{(-38)})$	Ten or eleven decimal digits accuracy with the same range as REAL.

SECTION III

Expressions and Assignment Statements

EXPRESSIONS

Expressions are combinations of primaries and arithmetic, logical, and relational operators. Primaries are constants, simple and array variables, function references, and parenthesized expressions of any type except character.

Expressions consist of three main types: arithmetic, logical, and character. Arithmetic expressions return a single value of type integer, real, double precision, or complex. Logical expressions evaluate to either .TRUE. or .FALSE. or to a 16-bit mask which can be used in later computations (depending on the context of the expression). Character expressions manipulate character primaries and return character values.

Arithmetic Expressions

A simple arithmetic expression is a primary (constant, variable, or function reference). These simple expressions can be joined together to form more complicated expressions using the following arithmetic operators:

+	Addition
–	Subtraction
*	Multiplication
/	Division
**	Exponentiation

For example,

```
6+ ABLE
(X + Y + Z) * 6
FOG ** X / (A + 6)
```

The hierarchy of arithmetic operations is

**	Exponentiation		
*	Multiplication and	/	Division
+	Addition and	-	Subtraction

-Division and multiplication occur before addition and subtraction within an expression, and exponentiation precedes all other operations. For example,

    A**B+C*D+6

is evaluated by

A**B = s_1(*s_1 and s_2 are intermediate results)*

C*D = s_2

$s_1 + s_2 + 6$ =the evaluated expression

Operators of the same class are evaluated according to the type of primaries involved in the operation, not necessarily from left to right.

Parentheses may be used to control the order of evaluation of expressions. For example,

    A + B + C

is evaluated according to the types assigned to A, B, and C. (The programmer does not control the order of evaluation.)

    A + (B + C)

evaluates (B + C) first, and then adds it to A while

    (A + B) + C

evaluates (A + B) first and then adds it to C.

    (A + B + C) - ((C + D) + X + Y)

In this expression, the evaluation of (A + B + C) occurs according to the variable types. In ((C + D) + X + Y) C + D is evaluated and then added to either X or Y depending on X's or Y's type. Finally, ((C + D) + X + Y) is subtracted from (A + B + C).

Two arithmetic operators cannot appear in a row. For example, A**-3 is illegal; A**(-3) is allowed.

Partial-Word Designators. A partial-word designator acts as a unary operator which extracts a specified bit string from the primary it suffixes and right-justifies the string to form a new value of the same type. This operator applies to integer primaries in arithmetic expressions, and to logical primaries in logical expressions. (The primary itself remains unchanged.) The partial-word designator form is

primary [*first bit*:*number of bits*]

primary	integer (or logical) constant, variable, function reference, or (expression).
first bit	a decimal integer constant specifying the beginning bit position of the bit string. The integer-word sign-bit is number 0, and the least significant bit of the integer word is bit 15.
number of bits	a decimal integer constant specifying the length of the bit-string (cannot exceed 15). If *number of bits* is not specified, the length is equal to (16 - *first bit*).

If *number of bits* is greater than (16 - *first bit*), the bit string wraps around (takes bits from bit 0, 1, etc.). For example, the partial-word designator [15:3] extracts bits 15, 0, and 1 in that order.

Example 1

```
     %037745 = 0 0 1 1 1 1 1 1 1 1  1  0  0  1  0  1  Binary
                                                       number

               0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  Bit position

%037745[15:3] = 0 0 0 0 0 0 0 0 0 0  0  0  0  1  0  0  New binary
                                                       number

               0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  Bit position
                                             |  |  |
                                            15  0  1  Former bit
                                                       position
```

Example 2

```
  I = 3000 10 = 0 0 0 0 1 0 1 1 1 0  1  0  1  1  1  0  Binary
                                                       number

               0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  Bit position

         I[6] = 0 0 0 0 0 0 1 1 1 0  1  0  1  1  1  0  New binary
                                                       number

               0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  Bit position
                           | | | |  |  |  |  |  |  |
                           6 7 8 9 10 11 12 13 14 15  Former bit
                                                       position
```

In Example 1, wrap-around occurred when *number of bits* was greater than (16 - *first bit*).

In Example 2, the number of bits was left out so the default value (16 - *first bit*) was used. Ten bits were extracted from the value of I and right-justified in the new integer storage word.

Arithmetic Expression Type. Integer, real, double precision, and complex primaries may be freely intermixed in an arithmetic expression. Before an arithmetic operation is performed, the lower type primary is converted to a higher type. The expression takes on the type of the highest type primary in the expression. Primary types rank from lowest to highest as

Integer

Real

Double precision real

Complex

Logical Expressions

A simple logical expression is a logical primary (constant, variable, function reference, (or expression), or two arithmetic expressions or character expressions joined by a relational operator (relation). Simple logical expressions are joined to form more complicated expressions using logical operators.

Relational Operations. The relational operators are

.EQ.	Equality
.NE.	Nonequality
.LT.	Less than
.LE.	Less than or equal
.GE.	Greater than or equal
.GT.	Greater than

These operators combine with arithmetic expressions or character expressions to form relations. Each relation is evaluated and assigned the logical value .TRUE. or .FALSE. depending on whether the relation between the two arithmetic expressions is satisfied (.TRUE.) or not (.FALSE.).

The two expressions in a relation must be of the same type: linear, complex, or character. A linear expression is an expression of type integer, real, or double precision; a complex expression is type complex and a character expression is type character. Complex expressions can be used as operands of .EQ. and .NE. only. The concept of "less than" or "greater than" is not defined for complex numbers.
For example,

```
CHAR .EQ. "END"

X + 6 .GT. VAL

I[7:4] .NE. 45
```

Logical Operators. The logical operators are

.NOT.	Complement
.AND.	AND
.XOR.	Exclusive OR
.OR.	Inclusive OR

The .NOT. operator takes the complement of the logical value of the primary or relation immediately following the .NOT. operator, for example, if A is a logical primary,

```
A = .TRUE.          .NOT. A = .FALSE.

A = .FALSE.         .NOT. A = .TRUE.
```

The .AND. operator returns a value of .TRUE. if, and only if, the logical expressions on both sides of .AND. are evaluated as .TRUE., for example, if A and B are logical expressions,

```
A = .TRUE.
B = .TRUE.          A .AND. B = .TRUE.

A = .TRUE.
B = .FALSE.         A .AND. B = .FALSE.

A = .FALSE.
B = .TRUE.          A .AND. B = .FALSE.

A = .FALSE.
B = .FALSE.         A .AND. B = .FALSE.
```

The .XOR. operator (exclusive OR) returns a value of .TRUE. if, and only if, one (but not both) of the logical expressions on either side of the .XOR. is .TRUE. For example, if A and B are logical expressions,

```
A = .TRUE.
B = .TRUE.          A .XOR. B = .FALSE.

A = .TRUE.
B = .FALSE.         A .XOR. B = .TRUE.

A = .FALSE.
B = .TRUE.          A .XOR. B = .TRUE.

A = .FALSE.
B = .FALSE.         A .XOR. B = .FALSE.
```

The .OR. operator (inclusive OR) returns a value of .TRUE. if one or both of the logical expressions on either side of the .OR. is .TRUE. For example, if A and B are logical expressions,

A = .TRUE.
B = .TRUE.
	A .OR. B = .TRUE.

A = .TRUE.
B = .FALSE.
	A .OR. B = .TRUE.

A = .FALSE.
B = .TRUE.
	A .OR. B = .TRUE.

A = .FALSE.
B = .FALSE.
	A .OR. B = .FALSE.

Logical Operator Hierarchy. The hierarchy of logical operations is

.NOT.	Complement
.AND.	AND
.XOR.	Exclusive OR
.OR.	Inclusive OR

.NOT. operations are performed before .AND. operations, and .OR. operations are performed after all other operations in a logical expression.

Example 1

If expressions A and B are both .TRUE., .NOT. A .AND. B evaulates .NOT. A (= .FALSE.) first, then evaluates .FALSE. .AND. B (= .FALSE. .AND. .TRUE. = .FALSE.).

Example 2

If expression A = .TRUE. and B = .FALSE., .NOT. B .XOR. A AND B evaluates .NOT. B (= .TRUE.) first, then evaluates A .AND. B (= .FALSE.) then evaluates .TRUE. .XOR. .FALSE. (= .TRUE.).

Parentheses can be used to direct the order of evaluation of a logical expression.

Example 3

If expression A and B = .TRUE., .NOT. (A .AND. B) evaluates A .AND. B (= .TRUE.) first, then evaluates .NOT. .TRUE. (= .FALSE.).

Without parentheses, the .NOT. operation would be performed first.

Example 4

If expression A = .TRUE., B = .TRUE., and C = .FALSE., (A .OR. B) .AND. C .OR. B evaulates A .OR. B (= .TRUE.) first, then evaluates .TRUE. .AND. C (= .FALSE.) then evaluates .FALSE. .OR. B (= .TRUE.).

Masking Operations. Besides evaluating .TRUE. or .FALSE. conditions, logical operators perform bit-by-bit operations on 16-bit logical values as follows:

.NOT.
.NOT. 0 = 1
.NOT. 1 = 0

For example,

A = 1110010100110011_2 .NOT. A = 0001101011001100_2

.AND.
0 .AND. 0 = 0
1 .AND. 0 = 0
0 .AND. 1 = 0
1 .AND. 1 = 1

A = 1011001000111011_2
B = 1111111111111111_2

$$\begin{array}{rr} \text{A .AND. B} = & 1011001000111011_2 \\ + & \underline{1111111111111111_2} \\ = & 1011001000111011_2 \end{array}$$

.XOR.
0 .XOR. 0 = 0
1 .XOR. 0 = 1
0 .XOR. 1 = 1
1 .XOR. 1 = 0

A = 1100110011001100_2
B = 1111000011110000_2

$$\begin{array}{rr} \text{A .XOR. B} = & 1100110011001100 \\ & \underline{1111000011110000} \\ = & 0011110000111100 \end{array}$$

.OR.
0 .OR. 0 = 0
1 .OR. 0 = 1
0 .OR. 1 = 1
1 .OR. 1 = 1

A = 0100110010110011_2
B = 1100110101110000_2

$$\begin{array}{rr} \text{A .OR. B} = & 0100110010110011 \\ & \underline{1100110101110000} \\ = & 1100110111110011 \end{array}$$

Partial-word designators can be used in logical expressions. For complete details see "Partial-word Designators" in this section.

Character Expressions

A character expression consists of either a character primary (constant, variable or function reference) or a character variable or function reference followed by a substring designator. No other operators are used in character expressions. Character expressions can be joined by relational operators to form relations used in logical expressions. (See "Logical Expressions" in this section.)

Character Primaries. A character value is treated as an array of characters, each character being one element of the array. If a character value contains 16 characters, the first character of the string is in position 1, and the last character is in position 16. Strings of characters and each element in the strings can be manipulated using a substring designator.

Substring Designators. A substring designator is a unary operator which extracts specified substrings of characters from a character value and creates a new value from the extracted substring. The substring designator form is

name [first character: number of characters]

name	a character variable or function reference. (Substring designators cannot be applied to character constants.)
first character	a linear expression specifying the beginning of the substring. This expression is evaluated as an integer and must range from 1 to the length of the character value.
number of characters	A linear expression specifying the length of the extracted substring. This expression is evaluated as an integer and must range from 0 to (length of value - *first character* + 1). If *number of characters* is not specified, the default value (length of value - *first character* + 1) is used.

Example 1

VAR1 = "THIS IS A STRING"

VAR1[3:7] = "IS IS A"

VAR1[3:7] *extracts the substring starting from the third character of the value (the letter I) and continuing for six more characters.*

VAR1 = "THIS IS A STRING"

INDEX = 6

VAR1[INDEX + 3] = "A STRING"

VAR1[INDEX + 3] *extracts the substring starting with the ninth character (the letter A) and ending with the last character of the value; number of characters was not specified, so the default value (length of value - first character* + 1) was used.

ASSIGNMENT STATEMENTS

An assignment statement has the form

name = expression

name	a variable or function reference of arithmetic, logical or character type
expression	an expression of the same type as *name*

When an assignment statement is executed, the expression is evaluated and the resulting value is assigned to the variable or function name.

Arithmetic expressions may be of type integer, real, double precision, or complex. The name need not be of the same arithmetic type as the expression. The value of the expression is converted to the name type before the value is assigned. (See Table 3-1.)

Partial-word designators can be used with logical and integer variables and function references in the *name* side of an assignment statement.

Substring designators can be used with character variables in the *name* side of a character assignment statement.

Character expression values are truncated if the defined character variable length is less than the expression value length. Character expression values are left-justified and padded with blanks on the right if the defined character variable length is larger than the expression value length.

Table 3-1. Conversion Between Types

Convert Integer to	Convert Real to	Convert Double Precision to	Convert Complex to
Real: 16-bit integer value is converted to 32-bit floating point value.	**Integer:** Real value is truncated to integer and stored in one 16-bit word	**Integer:** Double precision value is truncated to integer and stored in one 16-bit word.	**Integer:** Discard the imaginary part and truncate real part to 16-bit integer word.
Double Precision: 16-bit integer value is converted to 48-bit floating point value	**Double Precision:** Trailing zeros are added to right of fraction to create a 48-bit floating point value.	**Real:** Double precision value is truncated and stored in a 32-bit floating point value.	**Real:** Discard the imaginary part and maintain the real part.
Complex: Integer value is converted to real and stored in real part of complex value. Imaginary part is set to zero.	**Complex:** Taken as the real part of the complex umber with the imaginary part set to zero.	**Complex:** Truncated to real alue and taken as real part of complex number. Set imaginary part to zero.	**Double Precision:** Discard the imaginary part and convert real to double precision.

LABEL ASSIGNMENT STATEMENTS

A label assignment statement is used to assign label values to integer simple variables. The form is

ASSIGN *statement label* TO *variable*

statement label

variable

Integer simple variables have two separate values, one of type integer and one of the psuedotype "label." These two values are independent of each other and can exist simultaneously. The "label" value is referenced in only two FORTRAN statements. The label assignment statement assigns a "label" value to an integer simple variable, and the assigned GO TO statement (Section V) uses that value. All other references to the variable are to its integer value.

SECTION IV

Specification Statements

Specification statements define the characteristics of data used in programs. Specification statements are nonexecutable; when compiled, they do not produce instructions in the object program. Specification statements must appear before the first executable statement in each program unit.

ARRAY DECLARATORS

One function of several specification statements (DIMENSION, COMMON, and Type) is to define the number of values and the arrangement of the values in an array. This information is supplied to the program through an array declarator. The form of an array declarator is

name (b_1 ..., b_n)

name	the array variable name; the array type is determined through a Type statement or through the implicit typing convention of using the first character of the variable name as a letter that determines the variable type.

The list of integers (b_1, ..., b_n) are the array bounds. The array bounds indicate the number of dimensions of the array, and the maximum number of elements in each dimension. For example, the array declarator I(3, 4, 5) indicates a three-dimensional array of type integer. The maximum subscript (bound) allowed for each of the three dimensions is 3, 4, and 5 respectively.

The number of elements in an array are calculated by multiplying the array bounds. For example, I(3, 4, 5) implies that the array, I, contains 3 x 4 x 5 = 60 elements.

The number of words of memory needed to store an array is determined by the number of elements in the array and the array type. Integer and logical arrays store each element of the array in a single 16-bit computer word; real arrays store each element in two words; double precision real arrays store each element in three words; and complex arrays store each element in four words. If I (3, 4, 5) is type integer, it takes up 3 x 4 x 5 x 1 = 60 words of memory. Real array A(3, 4, 5) takes up 3 x 4 x 5 x 2 = 120 words of memory. A character array, CH(2, 2, 2) with elements consisting of three characters each (CHARACTER*3) takes up 2 x 2 x 2 x 3 = 24 contiguous characters in memory, packed two characters per word.

The array is stored as a one-dimensional array in memory according to the Array Successor Function, described under "*Equivalence Between Arrays of Different Dimensions*" in this section.

One, and only one, array declarator must occur for each array used in a program unit. The declarator can be contained in a DIMENSION, COMMON, or Type statement within the program unit. (See "Dimension Statements," "Common Statements," and "Type Statements" in this section.) If the array is used in a DIMENSION statement, the array name only (not the array declarator) can be used in a COMMON or Type statement. For example,

INTEGER ARR	Type statement specifying ARR as an integer name
DIMENSION ARR(4,4)	DIMENSION statement specifying number of elements, bounds, and dimensions in the array, using an array declarator.

If the array declarator is used in a COMMON or Type statement, the array must not be mentioned in a DIMENSION statement. The array declarator used in the Type or COMMON statement creates the necessary storage space in memory, just as if the array were mentioned in a DIMENSION statement, for example,

```
INTEGER ARR(4,4)
```

has the same effect as

```
INTEGER ARR

DIMENSION ARR (4, 4)
```

Adjustable Array Declarators

Normally, the array bounds, (b_1, ..., b_n), are specified by positive (greater than zero) integer constants. The array bounds are fixed by the value of the constants.

In some cases, it is possible to use adjustable array declarators. The array bounds are specified by integer simple variables instead of integer constants. Using adjustable bounds in the form of integer variables allows subprograms to use noncommon, program-local arrays whose sizes are dynamically determined during execution. Integer simple variables can be used as array bounds in a situation where the array declarator appears in a procedure, and the variables used to indicate the array bounds are dummy arguments. If an adjustable array name is used as a dummy argument the values assigned to the variables used as dimension bounds must not specify an array larger than the bounds of the actual array used when the procedure is executed. (See Section VI for a discussion of procedures and dummy arguments.)

Adjustable array declarators cannot be used in COMMON statements.

DIMENSION STATEMENTS

DIMENSION statements define the dimensions and bounds of arrays. The form is

DIMENSION *decl.*, ..., *decl.*

where *decl.* is an array declarator: *name* $(b_1, ..., b_n)$.

A DIMENSION statement is used to allocate storage space for the arrays specified in its array declarators (*decl.*). The DIMENSION statement need not be used to define all arrays in a program unit. An array declarator for each array used in a program unit must appear only once in the program (either in a DIMENSION, COMMON, or Type statement). Using an array declarator in a COMMON or Type statement is equivalent (in terms of memory space set aside for the array) to using the array declarator in a DIMENSION statement. For example,

INTEGER ARR	Type statement specifying ARR as type integer. The *name* of the array, not the array declarator, appears,
DIMENSION ARR(4,4)	DIMENSION statement using the array declarator ARR(4,4), which causes 16 words of memory to be set aside for the array element values.

These two program statements can be replaced by one statement if the array declarator instead of the array *name* is used in the Type statement:

INTEGER ARR(4,4)	Type statement specifies array ARR as type integer, and because of the array declarator, also sets aside 16 words in memory for the array. This one statement has the same effect as the two statements in the previous example.

COMMON STATEMENTS

The COMMON statement reserves a block of global storage space that can be referenced by several program units (such as a main program and one or more subroutines). The COMMON statement allows one program unit to store data in a nonlocal area which can be read, manipulated and stored by other program units. These areas of common information are specified in the form.

COMMON/*blockname*/*data element*, ..., *data element*/*blockname*/*data element*, ..., *data element*...

blockname	either null (specified by two slashes with no intervening non-blank characters //) or from 1 to 15 alphameric characters (the first one must be a letter). *blocknames* are used to identify different common blocks. Different program units reference the same common block by using the same *blockname* in their COMMON statements. The first *blockname* (including the two /'s) can be omitted if the user desires that the first common block in a list of blocks go unnamed. *blocknames* may be used as variable names (but not procedure names) within the same program unit. FORTRAN distinguishes between the name when used in a COMMON statement and when used as a variable.
data element	a simple variable, array name, or array declarator. Using an array name in a COMMON statement implies that the array declarator appears in a Type or DIMENSION statement somewhere in the same program unit.

The length of a common block is determined by the number and type of the *data elements* associated with that block. The *data elements* are stored contiguously within their block according to their listed order within the COMMON statement,

Statement	**Description**
INTEGER I(4)	Type statement indicating an integer array of four elements. (Area reserved for I = 4 words.)
REAL ARR	Type statement indicating ARR as type real. (Area reserved for ARR = 6 words.)
COMMON I,ARR(3)	COMMON statement. Array ARR has three elements. (Total common area = 10 words.)

Common block storage is allocated at the time the program is loaded into core for execution and is not local to any one program unit. No dummy variable name, function, subroutine name, or array with an adjustable array declarator or adjustable length character variable may be used in a COMMON statement, nor many any of these elements be put in a common block with an EQUIVALENCE statement. (See EQUIVALENCE in this section.) No name used in a DATA statement may be used in a COMMON statement or put in a common block through equivalence.

Data space within the common area is allocated as follows:

Word	Common Block
1	I(1)
2	I(2)
3	I(3)
4	I(4)
5	ARR(1)
6	ARR(1)
7	ARR(2)
8	ARR(2)
9	ARR(3)
10	ARR(3)

Correspondence of Common Blocks

Each program unit that uses the common block must include a COMMON statement which contains the *blockname* (if a name was defined). The *data elements* assigned to the common block by the program unit need not correspond by name, type, or number of elements. The only consideration is the original length of the common block as specified in the first COMMON statement mentioning that block. An unlabeled (unnamed) common block size may differ between program units, but a labeled common block must be the same size in all program units:

```
                                                Integer      Real
                                                   ↓           ↓
In program 1:        COMMON /BLOCKA/ I(4), J(6), ALPHA,SAM

                                             Real        Integer
                                               ↓            ↓   ↓
In program 2:        COMMON /BLOCKA/ GEO, L(10), INDIA,JACK
```

Thus, in the following example, referencing I(4) in program 1 is equivalent to referencing L(2) in program 2 since both variables pertain to the same word of the common block.

Program 1 Reference	Common Block Word Number	Program 2 Reference
I(1)	1	GEO
I(2)	2	GEO
I(3)	3	L(1)
I(4)	4	L(2)
J(1)	5	L(3)
J(2)	6	L(4)
J(3)	7	L(5)
J(4)	8	L(6)
J(5)	9	L(7)
J(6)	10	L(8)
ALPHA	11	L(9)
ALPHA	12	L(10)
SAM	13	INDIA
SAM	14	JACK

In the following example, the unlabeled common block is a different length in program 2 than in program 1. The last five words of the common block are not used by program 2.

```
In program 1:     COMMON I(12)

In program 2:     COMMON JAR(7)
```

Program 1 Reference	**Common Block Word Number**	**Program 2 Reference**
I(1)	1	JAR(1)
I(2)	2	JAR(2)
I(3)	3	JAR(3)
I(4)	4	JAR(4)
I(5)	5	JAR(5)
I(6)	6	JAR(6)
I(7)	7	JAR(7)
I(8)	8	Unused
I(9)	9	Unused
I(10)	10	Unused
I(11)	11	Unused
I(12)	12	Unused

If portions of a common block are not referred to by a particular program unit, dummy variables may be used to provide correspondence in reserved areas, for example,

```
                                      Integer
                                     ↙  ↓   ↘
In program 1:     COMMON /BLOCKA/ I(5), J(3), K(4)

                                Real Integer
                                  ↓       ↓
In program 2:     COMMON /BLOCKA/ ARR(4), K(4)
```

Program 1 Reference	**Common Block Word Number**	**Program 2 Reference**
I(1)	1	ARR(1)
I(2)	2	ARR(1)
I(3)	3	ARR(2)
I(4)	4	ARR(2)
I(5)	5	ARR(3)
J(1)	6	ARR(3)
J(2)	7	ARR(4)
J(3)	8	ARR(4)
K(1)	9	K(1)
K(2)	10	K(2)
K(3)	11	K(3)
K(4)	12	K(4)

ARR(4) is a dummy array which is never used by program 2 but provides proper correspondence so that program 2 can use array K.

Character Variables and Arrays in Common Blocks

A character variable occupies (character string length) x 1/2 words in the common block. For example, a character string of length 11 (11 characters) occupies five-and-one-half words.

A character array occupies the number of elements in the array times the length of each element times one-half. If an array contains 30 elements, and each element is five characters, the array occupies 75 words of the common block. Each data element in common starts at the next whole-word storage boundary following the preceding data element, except for string values, which start on the next half storage word. Thus,

```
INTEGER I(3), NO(3)

CHARACTER*3 CH(3)
.
.
.
COMMON I,CH,NO
```

In the following example the least significant half of word 8 is unused since the integer array NO starts on the first whole word boundary after the character array CH.

Program Reference	**Common Block Word Number**	
I(1)	1	
I(2)	2	
I(3)	3	
CH(1)	4	1 word
CH(1)	4	
CH(1)	5	1 word
CH(2)	5	
CH(2)	6	1 word
CH(2)	6	
CH(3)	7	1 word
CH(3)	7	
CH(3)	8	1 word
Unused	8	
NO(1)	9	
NO(2)	10	
NO(3)	11	

In the following example, the character array CH takes up five and one-half words in the common block. Character array BN starts in the least significant half of the fifth word (starts on a half-word boundary).

```
CHARACTER*3CH(3)
CHARACTER*1 BN(3)
.
.
.
COMMON CH,BN
```

Program Reference	**Common Block Word Number**	
CH(1)	1	1 word
CH(1)	1	
CH(1)	2	1 word
CH(2)	2	
CH(2)	3	1 word
CH(2)	3	
CH(3)	4	1 word
CH(3)	4	
CH(3)	5	1 word
BN(1)	5	
BN(2)	6	1 word
BN(3)	6	

EQUIVALENCE STATEMENT

The EQUIVALENCE statement associates simple variables and array elements so that they share all or part of their storage space. The form of the statement is

EQUIVALENCE (*list*), (*list*), ..., (*list*)

list	a data element list consisting of simple variables, array elements, or array names. Each *list* is enclosed in parentheses and separated by a comma. Each *list* indicates which variables and/or array elements (also separated by commas) are to share their storage space.

The following example of an EQUIVALENCE statement indicates that the named variables within the parentheses all share the same storage words in memory.

```
EQUIVALENCE (ABLE,SAM,ALPHA,DIX)
```

The statement below indicates that DIX and SAM share the same storage space, and that variables A and B share the same storage space.

```
EQUIVALENCE (DIX,SAM), (A,B)
```

Equivalence of Different Types

Equivalence between data elements of different types is allowable in FORTRAN, but care should be taken when attempting to match data types which store values in different size storage space. For example, if an integer and a real value are equivalenced, then the integer value is the same as the most significant word of the two-word real value:

Storage in Memory

REAL VALUE	Word 1	INTEGER VALUE
	Word 2	

Care should be taken when equivalencing character variables with other variable types. All data values other than character are stored in multiples of whole 16-bit computer words. Character values are stored in multiples of 8-bit half-words (two 8-bit characters per word). Character values may be equivalenced with other data types only if the resulting group can be allocated so that all noncharacter data elements begin on a whole-word boundary. For example,

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

CHARACTER*5 C(3)
```

Requires A and B to be allocated 5 half-words apart, which is illegal.

Equivalence of Array Elements

Array elements can be equivalenced to elements of a different array or to simple variables, for example,

```
DIMENSION A(3),C(5)

EQUIVALENCE(A(2),C(4))
```

In the preceding example array element A(2) shares the same storage space as array element C(4). This implies that

- A(1) shares storage space with C(3), and A(3) shares storage space with C(5).
- No equivalence occurs outside the bounds of any of the arrays.

The two statements below indicate that arrays A and C are type integer and that each arry has four elements and one dimension. C(1) and C(2) have unique store areas and A(3) and A(4) also have unique storage areas. A(1) shares space with C(3), and A(2) shares storage space with C(4):

```
    INTEGER A(4),C(4)

EQUIVALENCE (A(2),C(4))
```

Array A	**Storage Space Word Number**	**Array C**
	1	C(1)
	2	C(2)
A(1)	3	C(3)
A(2)	4	C(4)
A(3)	5	
A(4)	6	

Array elements are equivalenced on the basis of storage elements. If the arrays are not of the same type, they do not line up element by element. For example,

```
INTEGER A(4)

REAL B(2)

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

As shown below, A(1) and A(2) share the two computer words with the real array element B(1). A(3) and A(4) share the two computer words used to store the value of B(2).

Array A	**Storage Space Word Number**	**Array B**
A(1)	1	B(1)
A(2)	2	
A(3)	3	B(2)
A(4)	4	

Equivalence Between Arrays of Different Dimensions

To determine equivalence between arrays with different dimensions, FORTRAN provides an array successor function which views all elements of the array in linear sequence. This means that all arrays, regardless of their dimensions, are stored in memory as one-dimensional arrays. The following is a description of how the array successor function works. The right column shows how the array A(3,3,3) is viewed by the array successor function:

	Process	**Array Elements**
1.	The first element is designated by the subscript with all indexes equal to 1.	A(1,1,1)
2.	The next series of elements is determined by incrementing the leftmost index by one.	A(1,1,1) |A(2,1,1)

3.	Continue to increment the leftmost index by one until the index bound is reached.	A(1,1,1)
		A(2,1,1)
		|A(3,1,1)

4.	Once the index bound is reached, reset the index to 1 and increment the index immediately to the right by one.	A(1,1,1)
		A(2,1,1)
		A(3,1,1)
		|A(1,2,1)

5.	The next series of elements is determined by incrementing the leftmost index by one until the index bound is reached.	A(1,1,1)
		A(2,1,1)
		A(3,1,1)
		A(1,2,1)
		|A(2,2,1)
		|A(3,2,1)

6.	Once the index bound is reached, reset the index to 1 and increment the index immediately to the right by one.	A(1,1,1)
		A(2,1,1)
		A(3,1,1)
		A(1,2,1)
		A(2,2,1)
		A(3,2,1)
		|A(1,3,1)

	Process	Array Elements
7.	Determine the next series of elements by incrementing the leftmost index by one until the index bound is reached.	A(1,1,1)
		A(2,1,1)
		A(3,1,1)
		A(1,2,1)
		A(2,2,1)
		A(3,2,1)
		A(1,3,1)
		A(2,3,1)
		A(3,3,1)
8.	Both the leftmost and the index to the immediate right are at the index bound. Reset both indexes back to 1 and increment the third index by one.	A(1,1,1)
		A(2,1,1)
		A(3,1,1)
		A(1,2,1)
		A(2,2,1)
		A(3,2,1)
		A(1,3,1)
		A(2,3,1)
		A(3,3,1)
		A(1,1,2)
9.	This same process continues for each index. Generally, the next element is found by incrementing the leftmost index until it reaches its bound, then incrementing the next index by one only and returning all indexes to its left to one. The leftmost index is then incremented until it reaches its bound, etc.	A(1,1,1)
		A(2,1,1)
		A(3,1,1)
		A(1,2,1)
		A(2,2,1)
		A(3,2,1)
		A(1,3,1)
		A(2,3,1)
		A(3,3,1)
		A(1,1,2)
		A(2,1,2)
		A(3,1,2)
		A(1,2,2)
		A(2,2,2)
		A(3,2,2)
		A(1,3,2)
		A(2,3,2)
		A(3,3,2)
		A(1,1,3)
		A(2,1,3)
		A(3,1,3)
		A(1,2,3)
		A(2,2,3)
		A(3,2,3)
		A(1,3,3)
		A(2,3,3)
		A(3,3,3)

The following statements equivalence array elements A(2,2,2) and I(3). A is a three-dimensional array, and I is one-dimensional.

```
DIMENSION A(3,3,3),I(10)

INTEGER I,A

EQUIVALENCE (A(2,2,2), I(3))
```

The elements correspond as shown below.

Array A	Storage Word Relative Number	Array I
A(1,1,1)	1	
A(2,1,1)	2	
A(3,1,1)	3	
A(1,2,1)	4	
A(2,2,1)	5	
A(3,2,1)	6	
A(1,3,1)	7	
A(2,3,1)	8	
A(3,3,1)	9	
A(1,1,2)	10	
A(2,1,2)	11	
A(3,1,2)	12	I(1)
A(1,2,2)	13	I(2)
A(2,2,2)	14	I(3)
A(3,2,2)	15	I(4)
A(1,3,2)	16	I(5)
A(2,3,2)	17	I(6)
A(3,3,2)	18	I(7)
A(1,1,3)	19	I(8)
A(2,1,3)	20	I(9)
A(3,1,3)	21	I(10)
A(1,2,3)	22	
A(2,2,3)	23	
A(3,2,3)	24	
A(1,3,3)	25	
A(2,3,3)	26	
A(3,3,3)	27	

Equivalence in Common Blocks

Data elements may be put into a common block by specifying them as equivalent to data elements mentioned in a common statement. If one element of an array is equivalenced to a data element within a common block, the whole array is placed in the common block with equivalence maintained for storage units preceding and following the data element in common. The common block is always extended, if it is necessary to fit an equivalenced array into the block, but no array can be equivalenced into a common block if storage elements would have to be prefixed to the common block to contain the entire array. Equivalence cannot insert storage into the middle of the common block or rearrange storage within the block. Since elements in a common block are stored contiguously according to the order they are mentioned in the COMMON statement, two elements in common cannot be equivalenced. In the example below, array A is in a common block. Array element B(2) is equivalent to A(3).

```
DIMENSION B(6)

COMMON A(6)

EQUIVALENCE (A(3), B(2))
```

The common block is extended to accommodate array B as follows:

Array A	Common Block Word Number	Array B
A(1)	1	Not defined
A(2)	2	B(1)
A(3)	3	B(2)
A(4)	4	B(3)
A(5)	5	B(4)
A(6)	6	B(5)
Not defined	7	B(6)

The common block is extended by one word to store B(6).

The equivalence set up by the following statements is illegal. In order to set array B into the common block, an extra word must be inserted in front of the common block:

```
DIMENSION A(6),B(6)

COMMON A

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

Array A	Common Block Word Number	Array B
		B(1)
A(1)	1	B(2)
A(2)	2	B(3)
A(3)	3	B(4)
A(4)	4	B(5)
A(5)	5	B(6)
A(6)	6	Not defined

Element B(1) would be stored in front of the common block; EQUIVALENCE (A(1), B(2)) is not allowed.

Dummy variables, dummy array elements, function names, and subrouting names cannot occur in an EQUIVALENCE statement. A data element occurring in a DATA statement cannot be put into a common block through an EQUIVALENCE statement. None of the following elements can occur in an EQUIVALENCE statement:

- An array with an adjustable declarator.
- A character array of adjustable length.
- A character variable of adjustable length.

TYPE STATEMENTS

Type statements assign an explicit type to symbolic names representing variables and arrays (and their elements) and functions which would otherwise have their type implicitly determined by the first letter of their symbolic names. The Type statement form is

type element, element, element, ..., element, ...

type	consists of either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL or CHARACTER * *x* (*x* = the length of the character elements following)
element	consists of a simple variable name, array name, array declarator, function subprogram name, or a character simple name (optionally followed by * *x*)

If an array declarator is used in the element list, the declarator for that array must not be used in any other specification statement (such as DIMENSION or COMMON). If an array name is used, then an array declarator must appear within a DIMENSION or COMMON statement somewhere within the same program unit. The length of character symbolic names can be specified in two ways:

1. Through the length attribute following the CHARACTER heading (**x*), or

2. Through individual length attributes following character symbolic names

   For example, CHARACTER * 40 W,X,Y,Z defines the variable names W, X, Y, and Z as type character, each with a length of 40.

   The length attribute following the CHARACTER heading applies only to symbolic names not having their own length attribute. For example, CHARACTER * 30 X,Y,Z*20 defines two character variables of length 30, and one character variable Z of length 20. Character variables, array names, etc., are the only data elements which may have length attributes. Elements in a Type statement with any other heading cannot have length attributes.

   A CHARACTER heading or individual length attribute can use an adjustable length attribute in the form of an integer simple variable. For example, CHARACTER* (INT) X, Y, Z is a Type statement using an adjustable length attribute (INT, defined as an integer simple variable). The length attribute of X, Y, and Z depends upon the value assigned to INT.

   Adjustable length attributes in character Type statements can only be used in function subprograms and subroutine subprograms. The integer simple variable used as the length attribute must be a dummy parameter in the subprogram definition.

   A function of type CHARACTER cannot have an adjustable length.

A symbolic name can occur in only one Type statement within a program unit. If an intrinsic function name is used in a Type statement, the intrinsic function of that name is lost to the program unit. FORTRAN assumes that the name is defined within the program unit for some use other than that of an intrinsic function name.

Examples of Type statements are:

```
INTEGER A, B, C(6,6,6)

CHARACTER*25 A*6, FOUR*10(6,6), LEFT, RIGHT

COMPLEX CORE
```

IMPLICIT STATEMENT

The IMPLICIT statement reassigns the type associated with the initial letter of a symbolic name. If a symbolic name is not mentioned in a Type statement, the type of the data element is determined by the first letter of the symbolic name. Names starting with the I, J, K, L, M, or N are type integer, and names starting with any other letter are type real. The IMPLICIT statement rearranges this convention for any type desired (logical, integer, real, double precision, character, and complex). The IMPLICIT statement form is

IMPLICIT *type (letter, ..., letter), type (letter, ..., letter), ...*

type	INTEGER, LOGICAL, REAL, DOUBLE PRECISION, COMPLEX, or CHARACTER * *x* (* *x* is the length attribute of the character value, specified by a positive integer constant)
letter	a letter of the alphabet which assumes the type specified by the heading preceding it in the IMPLICIT statement; *letter* can be a single letter or a range of letters — for example, A–C means A,B,C.

The following statement specifies that all symbolic names starting with A, B, C, and E are type integer; D, F, G, and H are type real; J, K, and L are type complex; M, N, O, and P are double precision real and names starting with the letter Q are type character and have a length attribute of six characters.

```
IMPLICIT INTEGER (A - C,E), REAL (D,F - H), COMPLEX (J - L),

DOUBLE PRECISION (N - P,M), CHARACTER * 6 (Q)
```

In the CHARACTER heading, the length attribute (* *x*) cannot be specified by a variable. If the length attribute is omitted, the length is assumed to be one.

The implicit typing convention controlled by the IMPLICIT statement is overridden for specific symbolic names when these names are used in a TYPE statement. For example, IMPLICIT INTEGER (A) specifies that symbolic names starting with A are type integer. A Type statement such as REAL ABLE indicates that the variable ABLE is type real, overriding the IMPLICIT statement in that case. Only one IMPLICIT statement per program unit can be used.

EXTERNAL STATEMENTS

EXTERNAL statements identify function subprograms and subroutine subprograms which are called in one program unit but are defined in another (external) program unit. If a function subprogram or subroutine subprogram is used as actual argument, the function or subroutine name must be used in an EXTERNAL statement within the calling program unit.

The form of an EXTERNAL statement is

EXTERNAL *name*, *name*, ..., *name*, ...

where *name* is the symbolic name of a function or subroutine

Two examples are

```
EXTERNAL SUBRUT, ARRFIX

EXTERNAL ALPHA,BETA,DELTA
```

Functions and subroutines used as actual parameters in a function reference can be declared external to the program unit containing the function reference in two ways:

1. Mention the symbolic names of the functions or subroutines used as actual parameters in an EXTERNAL statement contained in the same program unit as the function reference, or

2. Suffix the symbolic names used as actual parameters with empty parentheses, for example,

   ```
   FUNC (NOWHERE ( ), SUBRUT ( ))
   ```

DATA STATEMENTS

DATA statements allocate local storage space and/or supply initial values for the data elements listed. The two forms of the DATA statement are

DATA *element, element, ..., element*

*DATA element, element, ..., element/*initial value list/, *element, element, ..., element/initial value list/, ...*

element	a simple variable name, array name, or array element
initial value list	a list of constants (separated by commas) including Hollerith constants

Allocation of storage space for elements of DATA statements can be reordered by EQUIVALENCE. Storage space is allocated statically; storage space for the data elements is set aside at the time the program unit containing the DATA statement is loaded and remains in memory even when the program unit is not executing.

The following format is legal:

DATA *element, element, ..., element/initial value list/, element, ..., element, ...*

For example,

```
DIMENSION IARR(4,4)

DATA IARR
```

results in

Data Elements	**Storage Word Number**
IARR(1,1)	1
. (2,1)	2
. (3,1)	3
. (4,1)	4
. (1,2)	5
. (2,2)	6
. (3,2)	7
. (4,2)	8
. (1,3)	9
. (2,3)	10
. (3,3)	11
. (4,3)	12
. (1,4)	13
. (2,4)	14
. (3,4)	15
. (4,4)	16

If the DATA statement has the form,

DATA *element, ..., element/initial value list/, element, ..., element/inital value list/ ...*

The constants in the *initial value list* are matched one-for-one to the simple variables and the array elements occuring in front of the initial value list. For example, the statement

```
DATA I, J, K /6,7,8/
```

assigns I=6, J=7, and K=8.

If a constant is specified one or more times in a row, the list may be abbreviated using a repeat factor. For example,

```
DATA A,B,C/ 3*0/
```

is the same as

```
DATA A,B,C /0,0,0/
```

Mentioning an array name is the same as mentioning all the elements of the array. For example, the statement

```
INTEGER A(3)

DATA A /3*0/
```

assigns A(1) = 0, A(2) = 0, and A(3) = 0. The statement

```
DATA A/3*0/
```

is the same as

```
DATA A(1),A(2),A(3)/0,0,0/.
```

The constants in the *initial value list* must be the same type as the data elements to which they are assigned[1]. Integer variables must be initialized by integer constants, real variables by real constants, and so on. A Hollerith or string constant can be used to initialize a data element of any type. For a character variable, the initial value represented by the Hollerith character is the character value itself. For any other type variable, the 8-bit ASCII patterns of the constant are stored left-justified in the storage space reserved for the variable. If the constant does not fill the entire storage space, the remaining part of the storage word is padded with the 8-bit ASCII code for blanks. Two characters are stored in one 16-bit computer word. For example,

```
DATA I,J,K,L/4*2HSA/
```

I, J, K, and L are integer variables. The ASCII 8-bit patterns for S and A are loaded into each 16-bit word for I, J, K, and L. It is not necessary to set initial values for all of the data elements listed in a DATA statement. For instance, the statements

```
DATA A / 10*8.0/

REAL A(20)
```

set up 20 elements of storage for array A, but only the first 10 of those elements are initialized to the value 8.0 (the other 10 elements of A are not given an initial value).

Equivalence in DATA Statements

Variables and array elements can be allocated in data blocks through the use of EQUIVALENCE statements. If an array element is equivalenced to an element in a DATA statement, the entire array is allocated in the data block. The data block is extended either at the beginning or the end to accommodate data elements set into the block through the EQUIVALENCE statement. For example,

```
INTEGER A(5), B(7)

DATA A

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

[1]A real variable can be initialized by an integer constant consisting of an optional sign followed by a string of digits. E.G., 20 can be used instead of "20.".

In the following example the data block is prefixed with one storage word to accommodate B(1) and extended one word to accommodate B(7). Note also that DATA A allocates five words of storage for array A. The EQUIVALENCE (A(1), B(2)) statement extends the data block one word in front and one word at the end to accommodate all the elements of array B.

Array A	Data Block Word Number	Array B
Unused	1	B(1)
A(1)	2	B(2)
A(2)	3	B(3)
A(3)	4	B(4)
A(4)	5	B(5)
A(5)	6	B(6)
Unused	7	B(7)

Equivalence can rearrange the order of storage allocation in the data block as long as all arrays remain contiguous within themselves. For example,

```
DIMENSION I(5), J(5), K(10)

DATA I,J

EQUIVALENCE (K(1), J(1)), (K(6), I(1))
```

is allowable and produces the result:

Array K	Data Block Word Number	Arrays I and J
K(1)	1	J(1)
K(2)	2	J(2)
K(3)	3	J(3)
K(4)	4	J(4)
K(5)	5	J(5)
K(6)	6	I(1)
K(7)	7	I(2)
K(8)	8	I(3)
K(9)	9	I(4)
K(10)	10	I(5)

No simple variable or array element can belong to both a common block and a data block, either explicitly through the use of DATA and COMMON statements, or implicitly through EQUIVALENCE statements. (See Block Data Subprograms for exceptions.) No dummy arguments or arrays with adjustable declarators can belong to a data block. EQUIVALENCE statements cannot be used if they try to store two elements of the same array into the same space in memory, or if they destroy the contiguity of the array elements. Noncharacter values will be stored starting on a full-word boundary in memory.

Block Data Subprograms

Block data subprograms exist for the sole purpose of supplying initial values to elements contained in common blocks. DATA statements are used in data block subprograms to supply these initial values. Storage space is allocated by the COMMON statements, and the initial values are supplied by the DATA statements. (For further discussion of block data subprograms, see Section VI.)

STATEMENT FUNCTIONS

A statement function is a computational procedure defined within the program unit that references it. The form of a statement function is

name (param, param, ..., param) = expression

name	a symbolic name starting with a nonnumeric character.
param	a simple variable used as a dummy argument. No other symbolic names except simple variable names may be used.
expression	an arithmetic or logical expression of constants, simple variables, array variables, function subprogram references, intrinsic references, statement function references, and the appropriate operators for the type expressions.

The statement function is defined in the program unit in which it will be used. The definition must occur before the first executable statement of the program unit and after all other specification statements (except for DATA statements). The statement function name may not be used in an EXTERNAL statement. The definition is a single statement similar to an arithmetic assignment statement.

The expression defines the actual computational procedure which derives one value. When referenced, this value is assigned to the function name. The expression must be either a logical expression or an arithmetic expression; no character expressions or character-valued function statements are allowed.

Any statement function referenced in the definition of another statement function must be defined before it is used in the definition. Statement function definitions are not recursive, that is, a statement function cannot reference itself.

The value of any dummy arguments in the expression are supplied at the time the statement function is referenced. (See Section II.) All other expression elements are local to the program containing the reference and derive their values from statements in the containing program.

The type of the statement function is determined by using the statement function name in a Type statement or by the first letter of the statement function name (names beginning with I, J, K, L, M, or N are type integer, while names beginning with all other letters are type real). This convention may be altered using the IMPLICIT statement.

The type of the expression must be compatible with the defined type of the statement function's symbolic name. Logical expressions must be used in logical statement functions and arithmetic expressions in arithmetic statement functions. The expression need not be the same arithmetic type as the statement function symbolic name. The expression value is converted to the statement function type at the time it is assigned to the statement function's symbolic name. (See Section III for a discussion of type conversion.) Two examples are

```
PROGRAM EX

REAL HOFFER

HOFFER (X,Y) = X ** Y + C
.
.
.
END
```

Note that the real statement function, HOFFER, is defined before the first executable statement of the program and after any other specification statements. In the following example the statement function uses a function subprogram reference as part of the defining expression.

```
PROGRAM WHY

EXTERNAL TOM

HARRY (X,Y,Z) = TOM (X,Y) + Z
.
.
.
END
```

SECTION V

Control Statements

Program execution normally proceeds sequentially from statement to statement. Control statements alter this sequence by transferring control to a specified statement or by repeating a predetermined group of statements.

Statements within a program unit are labeled by unsigned integers (1 through 99999). Embedded blanks and leading zeros in the label are ignored; 1, 01, 0 1, and 0001 are identical.

GO TO STATEMENTS

GO TO statements transfer control to the appropriately labeled statement in the same program unit. The three kinds of GO TO statements are unconditional, computed, and assigned.

Unconditional GO TO

The form of an unconditional GO TO statement is

GO TO *k*

where *k* is a statement label number. This statement transfers control to the statement labeled *k* every time it is executed. For example, when this statement is executed, control is transferred to statement 43 instead of the statement immediately following the GO TO statement.

20	GO TO 43
26	*(a statement)*
34	*(a statement)*
40	*(a statement)*
43	*(a statement)*
48	*(a statement)*

Computed GO TO

The form of a computed GO TO is

GO TO (*label, label, ..., label*), *index expression*

label	an unsigned integer from 1 to 99999.
index expression	an arithmetic expression of any type other than complex.

In a computed GO TO, the *index expression* is evaluated and converted to an integer value. (See "Arithmetic Expression Type," Section III.) The index is then used to pick one of the statement labels in the label list. Control is passed to the statement with that label. For instance, if the index is 1, the first label in the list is used; if the index is 2, the second is used; and so on. If the index evaluates to an integer less than 1, the first label in the list is used; and if the index evaulates to an integer greater than the positional number of the last label in the list, the last label is used. For example,

```
          I=0
20        I = I + 1
          GO TO (30,40,50), I
30        A = J + 1
          .
          .
          .
40        EXP = C * D
          .
          .
          .
50        KAY = 4 + JAY
```

In the preceding example I is the index expression. The first time the GO TO statement is encountered I = 1; thus control is transferred to statement 30. The next time the GO TO statement is executed, I = 2, thus control passes to statement 40. The third time, and any subsequent time the statement is executed, control is passed to statement 50 (as long as I remains greater than 3).

Assigned GO TO

The form of an assigned GO TO statement is

GO TO *variable*

or

GO TO *variable*, (*label*, *label*, ..., *label*)

variable	an integer simple variable.
label	an unsigned integer from 1 to 99999.

In either of the preceding forms, *variable* must be given a value through an ASSIGN statement prior to execution of the GO TO statement. Control is transferred to the statement whose label matches the value of *variable*. If *variable* is not given a label value or if the value given is not a valid existing label, an error message results.

The second from listed above includes a list of possible values that *variable* might take. The list is not functional; the value of *variable* is not checked against this *label* list. The list is there to remind the programmer of the possible places control might be transferred and is part of the documentation of the program.

In the following example, control is passed to statement 60 when the GO TO I statement is executed. I was assigned a valid statement label before the statement was executed.

```
10          ASSIGN 60 to I
            GO TO I
            .
            .
            .
60          JAY = KAY + ELL
```

IF STATEMENTS

IF statements control program flow in ways similar to GO TO statements. An arithmetic IF statement transfers control to one of three labeled statements depending whether the index expression is positive, negative, or zero. A logical IF statement defines an executable statement which is executed only if a conditional clause evaluates as true.

Arithmetic IF

The form of an arithmetic IF statement is:

IF (*expression*) *label, label, label*

expression	an arithmetic expression of any type except complex.
label	an unsigned integer from 1 to 99999.

In an arithmetic IF statement *expression* is evaluated. If the value is negative, control is passed to the statement whose *label* is first in the list. If the expression value is zero, control is passed to the statement whose *label* is second in the list. If the value is positive, then the last *label* in the list is chosen. Two or all of the labels in the list may be the same. For example,

```
            IF (I - 3) 30,40,50
            .
            .
            .
30          L = 7
40          K = 9
50          M = 11
```

If I is less than 3, control is transferred to statement 30; if I is equal to 3, control is transferred to statement 40; and if I is greater than 3, control is transferred to statement 50.

Logical IF

The form for a logical IF statement is

IF (*logical expression*) *statement*

logical expression	a logical expression as defined in Section III
statement	an input/output statement, an assignment statement, or a control statement other than a DO statement.

In a logical IF statement, the *statement* following the *logical expression* is executed as false, the statement is not executed. For example,

```
IF (A .EQ. 6) S = B + D
```

If A does equal 6, then the expression in parentheses evaluates as true, and S = B + D is executed. If A is not equal to 6, S = B + D is ignored. Thus, for example,

```
IF (A .EQ. 6) GO TO 40
```

If A equals 6, the unconditional GO TO statement is executed, and control is passed to the statement 40. If A does not equal 6, the GO TO is not executed and control passes to the statement immediately following the IF statement.

DO STATEMENTS

A DO statement controls execution of a predefined group of statements. The form of a DO statement is

DO *label variable* = *init, limit, step*

or

DO *label variable* = *init, limit*

label	the statement label for the last statement of the group controlled by the DO
variable	an integer simple variable which controls the number of times the group of statements is executed
init	the inital value given to *variable* at the start of execution of the DO statement
limit	the termination value for *variable*
step	the increment by which *variable* is changed after each execution of the group of statements defined by *label. Step* can be positive or negative.

init, limit, and *step* are indexing parameters. All three are arithmetic expressions of any type except complex, although their values are converted to integer whenever they are used by the DO mechanism. If *step* is omitted, it is assumed equal to 1. *init* and *limit* can be positive, negative, or zero.

Range and Execution of DO Loops

A DO statement defines a loop. The range of the DO loop is defined as the first statement following the DO statement, up to and including the terminal statement referenced by *label*. When the DO statement is executed, the following steps occur:

1. The control variable (*variable* in the DO statement) is assigned the value of *init*.

2. Control is passed to the first executable statement after the DO statement, and the range is executed.

3. The termination statement (defined by *label* in the DO statement) of the range is executed and *variable* is incremented by the value of *step*. If *step* is not mentioned in the DO statement, the control varaible is incremented by 1.

4. The control variable is compared with *limit*.

   - IF *step* is positive, the sequence is repeated starting at step 2 (if *variable* is less than or equal to *limit*). If *variable* exceeds *limit*, the DO loop is satisfied, and control transfers to the statement following the termination statement.

   - If *step* is negative, the sequence is repeated starting at step 2 (if *variable* is greater than or equal to *limit*). If *variable* drops below *limit*, the DO loop is satisfied, and control transfers to the statement following the termination statement.

Step 4 indicates that two possible cases exist when comparing the control variable with the limit parameter. When *step* is negative, the control variable must be less than *limit* before the DO loop passes control. When *step* is positive, the control variable must be greater than *limit* before the DO loop passes control. If either of the two cases exist when the loop is first entered, the instructions in the range of the DO loop are executed once only.

limit or *step* must not be redefined during execution of the range of the loop. *Variable* can be redefined during execution of the range of the loop.

The termination statement of a DO loop may not be a GO TO statement, arithmetic IF statement, RETURN statement, STOP statement, DO statement, or a logical IF statement which contains any of the previously mentioned statements.

Nesting DO Loops

If a DO loop is completely contained within the range of another DO loop, the first loop is nested within the second. The last statement of the nested loop (specified by the label in the DO statement) must either be the same as the terminating statement of the outer loop, or must occur before the outer loop terminating statement. For example,

```
 10     DO 100 I = 1,10,2
        .
        .
        .
 20     DO 90 J = 1,10,2
        .
        .
        .
 30     DO 80 K = 1,10,2
        .
        .
        .
 80     A = B + C
        .
        .
        .
 90     JAY = KAY + ELL
        .
        .
        .
100     X = C*D+6
```

The DO loop defined by statement 30 and 80 is nested within the DO loop defined by statements 20 and 90. These two DO loops are both nested within the range of the outer DO loop defined by statements 10 and 100. Thus, each DO loop has its own unique terminating statement. In the following example, the two outer loops have the same terminal statement.

```
 10     DO 100 I = 1,10,2
        .
        .
        .
 20     DO 100 J = 1,10,12
        .
        .
        .
 30     DO 90 K = 1,10,2
        .
        .
        .
 90     A = G * D
        .
        .
        .
100     Y = X + Z
```

The three DO loops are satisfied in the following manner:

1. The control variable I is initiated for the outer loop and control is passed to the statement following statement 10.

2. When statement 20 is executed, control variable J is initialized for the next inner loop and control is passed to the statement following statement 20.

3. When statement 30 is reached, control variable K is initialized for the innermost loop and control is passed to the statement following statement 30.

4. When statement 90 is reached, control is returned to the statement following 30. This continues until the innermost loop is satisfied.

5. Control is passed to the statement following 90.

6. When statement 100 is reached, control variable J is incremented and checked against the limit. Control is passed to the statement following 20. This continues until the "J" loop is satisfied.

7. When statement 100 is reached on the last pass through the J loop, the I control variable is incremented and checked, and control is returned to the statement after 10.

8. The two inner nested loops must again be satisfied before control is returned to the top of the outermost loop.

For example,

```
 10     DO 100 I = 1,10,2
        .
        .
        .
 20     DO 100 J = 1,10,2
        .
        .
        .
 30     DO 100 K = 1,10,2
        .
        .
        .
100     CONTINUE
```

If one or more loops have the same terminal statement, the control variable for the next outer loop is incremented and tested against the associated limit when the inner DO loop is satisfied. Control transfers to past the terminal statement only after all three loops are satisfied.

DO loops may be nested to as many levels as desired, as long as the ranges do not overlap. An example of overlapping ranges is

```
10        DO 100 I = 1,10,2
          .
          .
          .
20        DO 200 J = 1,10,2        ILLEGAL: THE RANGES OF THE
          .                        TWO LOOPS OVERLAP
          .
          .
100       X = Y
          .
          .
          .
200       GEO = A * B
```

Entering and Exiting DO Loops

A DO loop may be exited at any time, e.g., by a GO TO statement or a subprogram call, as long as the statement causing the passing of control is not the termination statement of the loop. For example,

```
10        DO 50 I = 1,10,2
          .
          .
          .
          GO TO 500
          .
          .
          .
50        CONTINUE
          .
          .
          .
500       X = Y + Z
```

In this example, control passes out of the DO loop by means of a GO TO statement.

It is possible to pass control into the range of a DO loop, but the results of this transfer are not defined unless certain conditions are met. Transfers into a DO range should occur only if a transfer out of that same DO loop had occurred previously. The following represents a legal transfer out of the range of a DO loop and back into the same range.

```
               PROGRAM EXAMPLE
               .
               .
               .
     10        DO 50 I = 1,10,2 ─────┐
               .                     │
               .                     │
               .                     │
     15        GO TO 70              │
               .                     │
               .                     │
               .                     │
     20        X = Y * V + R         │
               .                     │
               .                     │
               .                     │
     50        CONTINUE ─────────────┘
               .
               .
               .
     70        VAL = BAN + 6
               .
               .
               .
     90        GO TO 20
```

Instructions executed after a transfer out of a DO loop should not modify the control variable of that loop.

Instructions executed after a transfer out of a DO loop can include DO statements. However, the ranges of the DO statements must not contain any means for exiting and reentering the range before the DO loop is satisfied. Otherwise, the space in memory used for the original DO loop mechanism will be destroyed. The following is an example of an illegal transfer:

```
             PROGRAM EXAMPLE
             .
             .
             .
   10        DO 100 I = 1,10,2  ----------+
             .                            |
             .                            |
             .                            |
   20        GO TO 150                    |
             .                            |
             .                            |
             .                            |
   25        GO TO 170                    |
             .                            |
             .                            |
             .                            |
   30        X = T                        |
  100        CONTINUE  -------------------+
             .
             .
             .
  150        DO 250 J = 1,10,2  ----------+
             .                            |
             .                            |
             .                            |
  170        IF (A = 6.0) GO TO 30        |
             .                            |
             .                            |
             .                            |
  250        CONTINUE  -------------------+
```

The loop defined by statements 150 and 250 contain a possible transfer out of the range of the loop. When a transfer is made out of the first DO loop (through statement 20) and the second DO statement is executed, the first DO loop mechanism is altered. An attempt to reenter the first DO loop range might cause arbitrary results. Hence, transferring from the range of one DO loop and executing another DO statement is legal only if the second DO loop range does not contain possible exit points other than those that cause normal satisfaction of the DO loop.

If a terminal statement is the terminal statement for two or more nested DO loops, a transfer to the termination statement is a transfer to the innermost loop. For example,

```
10      DO 100 I = 1,10,2
        .
        .
        .
20      DO 100 J = 1,10,2
        .
        .
        .
30      DO 100 K = 1,10,1
        .
        .
        .
35      IF (K = 1) GO TO 150
        .
        .
        .
40      Z = C + X
        .
        .
        .
100     CONTINUE
        .
        .
        .
150     Z = N * B + 6
        .
        .
        .
200     GO TO 100
```

In the above example, program control runs from statements 10 to 20 to 30 to 35 to 150 to 200 to 100. When transfer to statement 100 occurs, FORTRAN assumes that the range of the innermost loop has been reached and checks the DO parameters against the control vaiable K.

A Transfer out of an inner level of several nested DO loops, and a subsequent transfer some time later into the range of the one of the outer DO loops is allowable as long as the previously discussed rules for the transfer are obeyed. For example,

```
 10     DO 70 I = 1,10,2
        .
        .
        .
 20     DO 60 J = 1,10,2
        .
        .
        .
 25     X = Y + Z
        .
        .
        .
 30     DO 50 K = 1,10,2
        .
        .
        .
 35     GO TO 100
        .
        .
        .
 50     CONTINUE
        .
        .
        .
 60     CONTINUE
        .
        .
        .
 70     CONTINUE
        .
        .
        .
100     N = T + R
        .
        .
        .
110     GO TO 25
```

CONTINUE STATEMENTS

A CONTINUE statement is used as the last statement in a DO loop that would otherwise end in a prohibited instruction such as a GO TO or logical IF statement. The form is

```
CONTINUE
```

If CONTINUE is used elsewhere in a program, it acts as a do-nothing instruction and control passes to the next executable statement. For example,

```
 10      DO 100 I = 1,10,2
         .
         .
         .
 20      X = 6
         .
         .
         .
         GO TO 20
100      CONTINUE
```

In the above example, the last useful statement of the DO loop is a GO TO statement, which is not allowed to be a terminal statement. The CONTINUE statement is used to terminate the loop.

BREAK STATEMENTS

Break statements consist of two forms, STOP and PAUSE. Execution of a STOP statement causes termination of program execution. The PAUSE statement causes a program break if the program is executing in interactive mode or a program termination if the program is executing in batch mode.

The form of a STOP statement is

STOP *integer*

The form of a PAUSE statement is

PAUSE *integer*

integer an unsigned integer used to identify the specific PAUSE or STOP.

CALL STATEMENTS

A program references a subroutine by executing a CALL statement. The CALL statement has the form:

CALL *name*

or

CALL *name (param, param, ..., param)*

or

CALL *name (param, param, ..., param, $label, ..., $label)*

name	identifies the symbolic name of the subroutine called. It must be identical to the name used in the SUBROUTINE statement that defines the subroutine.
param	an actual argument defined by the program unit containing the CALL statement. Actual arguments must agree in number, order, and type with the dummy arguments defined in the SUBROUTINE statement. The actual arguments may be constants, simple variables, array names, expressions, or procedure subprogram names.
label	a statement label (prefixed with a $).

The CALL statement transfers control to the subroutine. When the subroutine is executed, the actual arguments in the CALL statement are associated with their equivalent dummy arguments in the SUBROUTINE statement. The subroutine is then executed using the actual arguments. When a RETURN or END statement is executed (in the subroutine), control is returned to the statement following the CALL statement in the calling program unit. Control also can be returned to other statements in the calling program if a RETURN *n* statement is executed. (See "RETURN Statements" in this section.)

The CALL statement in the main program below is used to reference the subroutine defined after the calling program:

```
         PROGRAM SAM
         COMMON A(10), B(10)
         INTEGER J
         REAL A,B
         DO 10 L = 1,10                Calling program
10       B(L) = L
         J = 6
         CALL MULT(J)
         DO 20 M = 1,10
20       WRITE (6,200) A(M)
         END
         SUBROUTINE MULT(K)
         COMMON A(10), B(10)
         DO 10 I = 1,10                    Subroutine definition
10       A(I) = B(I) * K
         END
```

The main program first loads array B with integers from 1 to 10. Integer variable J is assigned the value 6. CALL MULT(J) transfers control from the main program to the subroutine. The integer variable J is substituted in the CALL statement for the integer variable K in the SUBROUTINE statement. The actual result of calling the subroutine is to multiply the number stored in each element of array B by 6 (J = 6) and store the result in the corresponding element of array A. J is substituted for K whenever K appears in this subroutine. When the END statement in the subroutine is executed, control is passed back to the statement following CALL MULT(J) in the main program. Since array A is accessible to both the subroutine and the main through COMMON statements, the main program can use the results of the subroutine execution.

RETURN STATEMENTS

RETURN statements transfer control from a subprogram back to the calling program unit. The statement form is

```
   RETURN
or
   RETURN n
```

where n is a positive integer constant or integer simple variable with positive value.

In a subroutine subprogram, executing a RETURN statement of the first form returns program control to the statement following the subroutine CALL statement in the calling program. For example,

```
         PROGRAM SAM                  SUBROUTINE DIX
         COMMON A,B,C
         X = 6                        COMMON A,B,C
         .                            .
         .                            .
         .                            .
40       CALL DIX
50       IF (A .LT. 3) Y = 1          RETURN
         .                            END
         .
         .
         END
```

Statement 40 transfers control to subroutine DIX. When the RETURN statement is executed, control is returned to statement 50 in the main program.

In a function subprogram, only the first form of the RETURN statement is allowed. When a RETURN statement is executed in a function, control is returned to the expression in the calling program which referenced the function. The value given the function name is used to continue evaluation of the referencing expression.
For example,

```
         PROGRAM HUFF                    FUNCTION IDIV(L,M)
         EXTERNAL IDIV                   IDIV = L/M
         .
         .                               RETURN
         .
         I = 8                           END
         J = 4
30       Y = 6 + IDIV(I,J)
         .
         .
         .
         END
```

The function reference in statement 30 passes control to function IDIV, which assigns a value to the function name. When the RETURN statement is executed, control is returned to statement 30 in the main program. The value assigned by IDIV is used to evaluate the expression. Control then passes to the statement following statement 30.

In subroutine subprograms the second form, RETURN *n*, can be used, where *n* is a positive integer constant or simple variable which acts as an index to choose a statement label from the list following the actual parameters of the CALL statement. Control is passed to the statement in the calling program that is prefixed by the chosen statement number:

```
         PROGRAM                         SUBROUTINE SUBR(X,Y,*, *, *,)
         EXTERNAL SUBR                   X = HOF + 3
         .                               Y = HOF/x
         .                               .
         .                               .
10       CALLSUBR(A,B,$20,$30,$40)       .
20       T = A + B                       I = 2
         .
         .                               RETURN I
         .
30       T = A * B                       END
         .
         .
         .
40       END
```

In the SUBROUTINE statement a list of asterisks follow the two dummy parameters to show that alternate return points exist. In the CALL statement in the main program, the asterisks are replaced by $20, $30, and $40. (The $ prefix is necessary to distinguish statement labels from integer constants.) Control is passed to the subroutine. When the RETURN I statement is executed, I = 2. The second label in the list is choosen. Control returns to statement 30 in the main program.

If the index is less than 1 or greater than the number of statement labels listed, a complier error results (when the index is a constant), and a run error results when the index is a simple variable.

SECTION VI

Main Programs and Subprograms

An executable FORTRAN program consists of program units made up of a main program and any necessary subprograms. Subprograms are subroutine subprograms, function subprograms or block data subprograms, written and compiled separately from the main program. Procedure subprograms are subroutines or functions (containing executable instructions). Block data subprograms do not contain executable instructions.

A main program does not require any other program to activate it, while subprograms depend upon other program units for activation. Procedure subprograms are referenced or called by a main program or another procedure subprogram. A procedure subprogram can also call itself as part of the defined computational process; i.e., a subprogram may be defined recursively. A calling program unit is a main program or subprogram that references or calls another subprogram.

STATEMENT ORDER IN PROGRAM UNITS

Categories of statements must appear in the same order within program units. In general, all specification statements must appear before the first executable statement in the program unit (see Table 6-1).

Table 6-1. Program Unit Statements

Statement Category	**Statement Examples**
Subprogram statements	SUBROUTINE FUNCTION BLOCK DATA PROGRAM
Implicit statements	IMPLICIT
Specification statements	DIMENSION COMMON EQUIVALENCE TYPE EXTERNAL
Data statements	DATA
Statement function definitions	FUNC(A,B) = A + B
Executable statements	Assignment statements control statements I/O statements
END line	END

FORMAT statements (not shown in Table 6-1) are nonexecutable but can appear anywhere in the program unit after the subprogram statement and before the END line. Specification statements can appear in any order after any IMPLICIT statement and before any DATA statements. DATA statements can appear anywhere after specification statements. Statement function definitions appear after the last nonexecutable statement and before the first executable statement. Statement function definitions may appear in any order within their group. However, any statement function referenced as part of another statement function definition must physically precede the former function's use in the definition.

END LINES

The very last line of any program unit (main or subprogram) must be an END line in the form

```
END
```

The END line signals the end of the program unit to the compiler. In a main program, END acts as a STOP statement causing termination of execution. In a procedure subprogram, END acts as a RETURN statement if the END statement is executed. The END line should be prefixed by a label if it is intended to double as a RETURN or STOP statement.

MAIN PROGRAMS

A main program consists of any necessary nonexecutable instructions (IMPLICIT, specifications statements, or DATA statements, in addition to statement function definitions); one or more assignment, control, or input/output statements; and an END line. The main program can be assigned a symbolic name by using a PROGRAM statement as the very first statement of the program. The PROGRAM statement has the form

PROGRAM *name*

where *name* is an alphameric string from one to fifteen characters (the first character must be alphabetic).

Any main program not headed by a PROGRAM statement is assigned the special name MAIN' by the FORTRAN compiler.

SUBROUTINE SUBPROGRAMS

A subroutine is a computational procedure. A subroutine also can return values through actual arguments supplied by the calling program unit or through common storage. No value or type is associated with the name of the subroutine.

The first statement of a subroutine must be a SUBROUTINE statement, which gives the subroutine name and its dummy arguments, if any:

SUBROUTINE *name*

or

SUBROUTINE *name (param, param, ..., param)*

or

SUBROUTINE *name (param, param, ..., param, *, ..., *)*

name	alphameric string from one to fifteen characters (the first character must be alphabetic).
param	a dummy argument of the subroutine. *param* can be a simple variable, array name, subroutine name, or function subprogram name.
*	indicates that statement label (prefixed by a $) occurred in CALL statement for this subroutine.

The subroutine can define or modify any of its value-possessing arguments and common areas to return values to the calling program. The subroutine can contain any statement except another SUBROUTINE, FUNCTION, PROGRAM, BLOCK DATA statement. The subroutine can contain a CALL statement referencing itself (recursive call). Other statement categories must appear as defined earlier in Section VI.

The last line of a subroutine must be an END line. One or more RETURN statement can be included to return control to the calling program unit. If no RETURN statement exists, the END line acts to return control to the calling program unit. For example,

```
SUBROUTINE STORCOM

COMMON A(50)

REAL A

DO 10 I = 1,50

A(I) = A(I) * 6

RETURN

END
```

In this example, the subroutine named STORCOM multiplies each element of array A by 6. If it contains a COMMON statement, array A is stored in a common block and is therefore accessible to the calling program.

In the following example, the subroutine named FLAG compares two real numbers and assigns the value true to the logical variable SET if the numbers are equal; FLAG assigns the value false if the numbers are unequal. These values are stored in a common block so that they are accessible to both the subroutine and the calling program unit.

```
SUBROUTINE FLAG

COMMON SET, A,B

LOGICAL SET

REAL A,B

IF (A .EQ. B) SET = .TRUE.

IF (A .NE. B) SET = .FALSE.

END
```

Subroutines are invoked by using the subroutine name in a CALL statement (see Section V).

FUNCTION SUBPROGRAMS

A function subprogram is a computational procedure which returns a value associated with the function name. The first statement of a function must be

FUNCTION *name (param, param, ..., param)*

or

type FUNCTION *name (param, param, ..., param)*

name	an alphameric string from one to fifteen characters (the first character must be alphabetic).
param	a dummy argument of the function. It can be a simple variable, array name, subroutine name, or a function subprogram name.
type	either LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX or CHARACTER **n* (where *n* is a positive integer constant specifying the length of the character function value. *n* cannot be a variable. **n* can be omitted).

The type associated with the function name is determined in one of three ways:

1. If the type is mentioned as the first part of the FUNCTION statement, the function name is assigned that type.

2. If the type is not given in the FUNCTION statement, the function name can be mentioned in a Type statement within the calling program unit.

3. If the function name is not mentioned in a Type statement or the type mentioned in the FUNCTION statement itself, the type is assigned implicitly according to the first letter of the name. Names starting with I, J, K, L, M, or N are type integer, and names starting with any other letter are type real. (This convention may be modified by an IMPLICIT statement, see Section IV.)

To associate a value with the function subprogram name, the name must be used within the function subprogram as a simple variable in one or more of the following contexts:

1. The left side of an assignment statement

2. An element of an input list in a READ statement

3. An actual parameter of a function or subroutine

The value last assigned to the name of the function at the time a RETURN or END statement is executed within the subprogram is considered the value of that function. The body of the function may contain a reference to itself (recursive call).

In the following example, the function name is used as a simple variable and is given its value through assignment:

```
INTEGER FUNCTION DIVD(I,J)
DIVD = I/J
END
```

The function named DIVD is defined as type integer by the FUNCTION statement. The value of the function is determined through assignment. If I and J are both integers, I/J is always evaluated as an integer; i.e., 4/3 = 1, 7/2 = 3, 9/4 = 2.

The function name in the following example is given a value by using it as an input item in a READ statement:

```
REAL FUNCTION GETVAL
READ(5) GETVAL
END
```

GETVAL is associated as type REAL by the FUNCTION statement. The value of the function is determined by reading from an input device.

```
FUNCTION SCALL (A,B,C)

A = 6
.
.
.
CALL SUBF(SCALL, A,B)

RETURN

END


SUBROUTINE SUBF(X,S,T)

X = S+T

END
```

SCALL is the function name used as an actual parameter of the subroutine SUBF. SCALL is given a value through SUBF and this value is passed back to the calling program.

BLOCK DATA SUBPROGRAMS

Block data subprograms provide initial values for simple variables and array elements in labeled common blocks. A block data subprogram consists of a BLOCK DATA statement and IMPLICIT, COMMON, DIMENSION, EQUIVALENCE, Type, and DATA statements. EXTERNAL statements are not allowed in block data subprograms. The subprogram's last line must be an END line.

The first statement of a block data subprogram must be a BLOCK DATA statement:

```
    BLOCK DATA
or
    BLOCK DATA name
```

where *name* is a character string from 1 to 15 characters (the first character is alphabetic. The *name* may be included to identify the subprogram.

Block data subprograms do not generate code; they use DATA statements to supply initial values to variables in labeled common blocks. The common blocks must be fully specified in a COMMON statement. EQUIVALENCE, DIMENSION, and Type statements also can be used for defining the variables in the common blocks. The DATA statements indicate which variables mentioned in the COMMON statement have initial values and what those values are. No variable should be mentioned in a DATA statement in a BLOCK data subprogram unless it is mentioned in a COMMON statement. However, not all variables mentioned in a COMMON statement need be mentioned in a DATA statement—only those data elements which are to have initial values. DATA statements do not affect the storage allocation of any of the variables in block data subprograms.

More than one common block can be initialized in a single block data subprogram; for example,

```
    BLOCK DATA BL1

    COMMON /COMA/A,B,C/COMB/D,E,F

    REAL A,B,C,D,E,F

    DIMENSION A(20)

    DATA A,C/20 * 1.0,34.0/, E,F/ -4.3,67.9/

    END
```

The preceding block data subprogram describes two common blocks, COMA and COMB. COMA contains a real array of 20 elements called A, and two simple real variables called B and C. COMB contains simple real variables D, E, and F. The DATA statement supplies initial values for all 20 elements of array A, variable C, E, and F. Initial values are not supplied for B or D, even though they are both mentioned in the COMMON statement.

NON-FORTRAN LANGUAGE SUBPROGRAMS

Procedure subprograms written in a language other than FORTRAN can be used as long as the calling sequence and the effect of execution are consistent with FORTRAN. For details on the use of non-FORTRAN language subprograms, consult Appendix A.

SECTION VII

Functions

A function is a computational process which returns a single value to the function name of the type assigned to the name. Functions are either locally defined or globally defined. Locally defined functions are recognized only in the program unit which defines them. Globally defined functions are recognized in any program unit which declares the function name as EXTERNAL. Intrinsic functions and statement functions are locally defined functions, and function subprograms (including a distinguished set of function subprograms called basic external functions) are globally defined functions.

INTRINSIC FUNCTIONS

The symbolic names of the intrinsic functions are predefined to the FORTRAN compiler. Intrinsics can be used merely by writing a function reference with the appropriate actual arguments (see "Function References" in Section II. The symbolic name of an intrinsic function can be redefined within a program unit by using the name in a specification statement other than a Type statement of the intrinsic's normal type, or as a statement function or as a simple variable. The intrinsic name can also be redefined in a subprogram statement (SUBROUTINE, FUNCTION, BLOCK DATA). The new usage applies only to the program unit in which the redefinition is made. For a subprogram definition, the new usage applies in any other program unit where the intrinsic name is used in an EXTERNAL statement or in a Type statement different from the intrinsic's type. Otherwise, the intrinsic itself is invoked when the name is used in a function reference. Table 7-1 describes the intrinsics, their function references and their argument characteristics.

Table 7-1. Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Function Reference	Type of Argument	Type of Function
Absolute Value	$\|a\|$	1	ABS(a)	Real	Real
			IABS(a)	Integer	Integer
			DABS(a)	Double	Double[1]
Truncation	Sign of a times largest integer $\leq \|a\|$	1	AINT(a)	Real	Real
			INT(a)	Real Logical	Integer
			IDINT(a)	Double	Integer
			DDINT(a)	Double	Double
Remaindering[2]	a_1 (mod a_2)	2	AMOD (a_1, a_2)	Real	Real
			MOD (a_1, a_2)	Integer	Integer
Choosing largest value	Max (a_1, a_2, ...)	2	AMAX0 (a_1, a_2, ..., a_n)	Integer	Real
			AMAX1 (a_1, a_2, ..., a_n)	Real	Real
			MAX0 (a_1, a_2, ..., a_n)	Integer	Integer
			MAX1 (a_1, a_2, ..., a_n)	Real	Integer
			DMAX1 (a_1, a_2, ..., a_n)	Double	Double
Choosing smallest value	Min(a_1, a_2, ...)	2	AMIN0 (a_1, a_2, ..., a_n)	Integer	Real
			AMIN1 (a_1, a_2, ..., a_n)	Real	Real
			MIN0 (a_1, a_2, ..., a_n)	Integer	Integer
			MIN1 (a_1, a_2, ..., a_n)	Real	Integer
			DMIN1 (a_1, a_2, ..., a_n)	Double	Double

[1]Double = double precision real.

[2]The function MOD or AMOD (a_1, a_2) is defined as $a_1 - |a_1/a_2| a_2$, where $|x|$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

Table 7-1. Intrinsic Functions (Continued)

Intrinsic Function	Definition	Number of Arguments	Function Reference	Type of Argument	Type of Function
Float	Conversion from integer to real	1	FLOAT (a)	Integer	Real
Fix	Conversion from real to integer	1	IFIX(a)	Real	Integer
Transfer of sign	Sign of a_2 times $\|a_1\|$	2	SIGN(a_1, a_2)	Real	Real
			ISIGN(a_1, a_2)	Integer	Integer
			DSIGN(a_1, a_2)	Double	Double
Positive Difference	a_1 - Min(a_1, a_2)	2	DIM(a_1, a_2)	Real	Real
			IDIM(a_1, a_2)	Integer	Integer
Obtain most significant part of double precision argument		1	SNGL(a)	Double	Real
Obtain real part of complex argument		1	REAL(a)	Complex	Real
Obtain imaginary part of complex argument		1	AIMAG(a)	Complex	Real
Express single precision argument in double precision form		1	DBLE(a)	Real	Double
Express two real arguments in complex form	$a_1 + a_2\sqrt{-1}$	2	CMPLX(a_1, a_2)	Real	Complex

Table 7-1. Intrinsic Functions (Continued)

Intrinsic Function	Definition	Number of Argument	Function Reference	Type of Argument	Type of Function
Obtain conjugate of a complex argument		1	CONJG(*a*)	Complex	Complex
Obtain the position of the character in the first argument which begins the substring which matches the second argument	INDEX	2	INDEX(a_1, a_2)	Character	Integer
Convert character expression to integer (INUM), real (RNUM) or double-precision (DNUM)	INUM	1	INUM(*a*)	Character	Integer
	RNUM	1	RNUM(*a*)	Character	Real
	DNUM	1	DNUM(*a*)	Character	Double
Convert an arithmetic expression of any type except complex (a_1) string of length a_2.	STR	2	STR(a_1, a_2)	Integer, real, or Double	Character
Convert integer expression to type logical	BOOL		BOOL(*a*)	Integer	Logical

STATEMENT FUNCTIONS

Statement functions are similar in effect to intrinsic functions, recognized only within the program unit which defines it. Use of a statement function name in any program unit outside the defining one is purely local to the program unit outside the defining program unit. (See "Function References" in Section II, and "Statement Functions" in Section IV.)

In the following example the statement function (TWO) includes a reference to statement function ONE in its definition. This reference is allowable since ONE is defined before TWO in the program.

```
          PROGRAM EX

          ONE(X,Y) = (X + Y)**2        (Statement function definition)

          TWO(S,T) = (ONE(S,T))**2   (Statement function definition)
          .
          .
          .
          A = -1

          B = 3
          .
          .
          .
   20     Z = TWO(A + 2, B - 2)
          .
          .
          .
          END
```

The value of Z in statement 20 is calculated as

```
A + 2 = 1

B - 2 = 1

ONE(A + 2, B - 2) = ONE(1,1) = 2**2 = 4

TWO(1,1) = (ONE(1,1))**2 = 4**2 = 16

Z = 16
```

FUNCTION SUBPROGRAMS

Function subprograms are globally defined computational procedures. A reference to a function subprogram can appear in any program unit. (See "Function References" in Section II and "Main Programs and Subprograms" in Section IV.)

In the following example, program EX references the function LARGE to assign a value to the variable, Z.

```
FUNCTION LARGE(A,B)

REAL A,B

IF (A .LT. B) LARGE = B

IF (A .GT. B) LARGE = A

END


PROGRAM EX

REAL X,Y
.
.
.
Z X = LARGE(X,Y)
.
.
.
END
```

BASIC EXTERNAL FUNCTIONS

Some basic computational procedures (such as taking the square root of a number) are defined in FORTRAN as basic external functions. To use these functions, the function reference, along with the appropriate actual arguments, must appear in an expression.

The type of the function (if other than Integer or Real) must be defined in the user's program through a Type statement or use of an IMPLICIT statement. When the basic external function is referenced, the actual arguments are checked for proper type. The defined function type is associated with the results.

The user can define a function subprogram with the same name as a basic external function. The new function takes the place of the system defined function.

The defined basic external functions available are shown in Table 7-2. For complete details of basic external function see *HP 3000 Compiler Library (03000-90009).*

Table 7-2. Basic External Functions

Basic External Function	Definition	Number of Arguments	Function Reference	Type of Argument	Type of Function
Exponential	e^a	1	EXP(a)	Real	Real
		1	DEXP(a)	Double	Double
		1	CEXP(a)	Complex	Complex
Natural logarithm	$\text{Log}_e\ (a)$	1	ALOG(a)	Real	Real
		1	DLOG(a)	Double	Double
		1	CLOG(a)	Complex	Complex
Common Logarithm	$\text{Log}_{10}\ (a)$	1	ALOG10(a)	Real	Real
		1	DLOG10(a)	Double	Double
Trigonometric Sine	Sin (a)	1	SIN(a)	Real	Real
		1	DSIN(a)	Double	Double
		1	CSIN(a)	Complex	Complex
Trigonometric cosine	Cos (a)	1	COS(a)	Real	Real
		1	DCOS(a)	Double	Double
		1	CCOS(a)	Complex	Complex
Trigonometric tangent	Tan (a)	1	TAN(a)	Real	Real
Hyperbolic tangent	Tanh (a)	1	TANH(a)	Real	Real
Square root	$(a)^{1/2}$	1	SQRT(a)	Real	Real
		1	DSQRT(a)	Double	Double
		1	CSQRT(a)	Complex	Complex
Arctangent	Arctan (a)	1	ATAN(a)	Real	Real
		1	DATAN(a)	Double	Double
	Arctan (a_1/a_2)	2	ATAN2(a_1, a_2)	Real	Real
		2	DATAN2(a_1, a_2)	Double	Double
Remaindering[2]	a_1 (mod a_2)	2	DMOD(a_1/a_2)	Double	Double
Modulus		1	CABS(a)	Complex	Real

[1]Double = double precision real.

[2]The function MOD or AMOD (a_1, a_2) is defined as $a_1 - |a_1/a_2|a_2$, where $|x|$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

SECTION VIII

Input/Output Statements

Input/output statements transfer information between data elements in memory and external devices or between data elements in memory and user-defined buffers in memory.

An input/output statement can contain a list of names of simple variables, array elements, arrays, and function subprograms. When an input statement is executed, the input values from the external device or user-buffer are assigned to the symbolic names in the list. When an output statement is executed, the values assigned to the listed variables are transferred to the external device or to the specified buffer space in memory.

To reference a specific external device, an input or output statement references the file number associated with the device. The file facility transfers data to and from the external devices. (See Section X for a discussion of the FORTRAN/3000 file facility.)

Data is input or output in groups called records. These records may be formatted or unformatted. To transfer formatted data, a string of characters known as format and edit specifications must be used to convert the data to and from memory.

All input/output is handled through the FORTRAN Formatter program. For a complete description of the format and edit specifications and the interaction between the user program and the FORTRAN Formatter, consult Section IX.

READ STATEMENTS

Read statements transfer information from an external device or character buffer in memory to specified data elements in a list. The form of a READ statement is

READ (*control part*) *element, element, ..., element*

or

READ (*control part*)

where *control part* consists of

(unit, format, labels)

or

(unit, format)

or

(unit, labels)

or

(unit)

element is a simple variable name, array or array element name, function *subprogram name or DO-implied list (see below).*

Control Part

The control part of a READ statement consists of a unit reference, format reference, or action label reference in any combination specified above.

UNIT REFERENCE. *unit* specifies the source from which the data values are to be read (either a file associated with an external device, or a user-defined buffer in core).

If the source is a file, *unit* consists of

file	*(sequential access)*
file @ record	*(direct access)*
file	is a positive integer constant or integer simple variable which indicates the desired file number (between 1 and 99, inclusive).
record	is an arithmetic expression of any type except complex, This value is converted to type integer and indicates a specific record of a direct-access file. The @ must be used to separate the file number from the record expression.

If the source is a buffer in memory, unit consists of

name

where *name* is a simple character or character array element specifying the buffer which contains the data to be transferred.

FORMAT REFERENCE. *format* specifies the location of the format and/or edit specifications which determine how the data is to be converted. *format* consists of

statement label

or

array name

or

character variable name *(Simple or subscripted)*

or

*

where *statement label* indicates that format and edit specifications appear in the FORMAT statement prefixed by statement label, and *array name* or *character variable name* indicates that format and edit specifications are contained in the array or variable specified. An asterisk (*) indicates that data is in free-field format and does not require any format or edit specifications other than those already contained within the data to be transmitted.

If *format* is omitted in the READ statement *control part*, a binary transfer takes place.

ACTION LABEL REFERENCES. *labels* allow program control over exceptional conditions and take the form

END = *statement label*

or

ERR = *statement label*

or

END = *statement label,* ERR = *statement label*

or

ERR = *statement label,* END = *statement label*

END = *statement label* transfers program control to the statement identified by *statement label* if an end-of-file condition occurs (insufficient records, record too short for a binary read, etc.). ERR = *statement label* transfers control to the statement shown if a transmission error occurs (parity error, incorrect type, etc.).

If the action labels are left out of a READ statement, and an exceptional condition occurs, the FORTRAN/3000 program is terminated with the appropriate diagnostic message. For example,

```
            PROGRAM EXRE
            INTEGER A,B
            READ (3   3,210,END = 250,ERR = 260) A,B
    210     FORMAT ...
            .
            .
            .
    250     CALL ENDOF
            .
            .
            .
    260     CALL ERROF
            .
            .
            .
```

program EXRE reads two values from a file and assigns them to the integer variables A and B. The data read is converted through a FORMAT statement (labeled 210) containing format specifications. If an end-or-file condition occurs, program control is passed to statement 250, a call to subroutine ENDOF to process the end-of-file condition. If a data transmission error occurs, control is passed to statement 260, which calls for another subroutine (ERROF) to process the error condition.

READ STATEMENT EXECUTION. When the READ statement is executed, data values are transferred from the source indicated by *unit* in the READ statement *(control part)* to the data elements specified in the element list. Elements are assigned values left-to-right as they appear in the list.

Each READ statement begins reading values from a fresh record of the file, ignoring any values left unread in records accessed by previous READ statements. If a READ statement does not contain any elements, the "next record" pointer advances one record and no transfer of data takes place.

If the READ statement contains no format reference in the *control part,* a binary read is initiated. For a sequential file *(unit = file),* records are read sequentially until the last element in the element list receives a value. For a direct access file *(unit = file record),* only one record is read. The record must contain enough values so that all the elements in the element list receive a value. Otherwise, a run error occurs.

If the READ statement does contain a format reference in the *control part,* a formatted read is initiated. Records are read sequentially until all of the list elements have received a value, regardless of whether the file is direct or sequential access.

Array names appearing in an element list stand for all the elements of the array. Values are transferred to the array elements in the order prescribed by the array successor function (see Section IV).

ACCEPT STATEMENTS

An ACCEPT statement is a read statement intended for (but not restricted to) programs operating from a terminal device. The form is

ACCEPT *element, element, ..., element*

where *element* is a simple variable name, array name, array element, function subprogram name, or DO-implied list. (See "DO-implied Lists" in this section.)

When the ACCEPT statement is executed, it prints a question mark on the standard output device, e.g., a teleprinter. It then performs a free-field read from the standard input device (which may also be the teleprinter). For example,

```
PROGRAM RL
INTEGER A, B, C
ACCEPT A,B,C
END
```

When the program executes, it types the message ?. The user answers, for example, 35,455,733. This assigns the value 35 to A, 455 to B and 733 to C.

STATEMENT FUNCTIONS

Statement functions are similar in effect to intrinsic functions, recognized only within the program unit which defines it. Use of a statement function name in any program unit outside the defining one is purely local to the program unit outside the defining program unit. (See "Function References" in Section II, and "Statement Functions" in Section IV.)

In the following example the statement function (TWO) includes a reference to statement function ONE in its definition. This reference is allowable since ONE is defined before TWO in the program.

```
          PROGRAM EX

          ONE(X,Y) = (X + Y)**2       (Statement function definition)

          TWO(S,T) = (ONE(S,T))**2   (Statement function definition)
          .
          .
          .
          A = -1

          B = 3
          .
          .
          .
  20      Z = TWO(A + 2, B - 2)
          .
          .
          .
          END
```

The value of Z in statement 20 is calculated as

A + 2 = 1

B - 2 = 1

ONE(A + 2, B - 2) = ONE(1,1) = 2**2 = 4

TWO(1,1) = (ONE(1,1))**2 = 4**2 = 16

Z = 16

FUNCTION SUBPROGRAMS

Function subprograms are globally defined computational procedures. A reference to a function subprogram can appear in any program unit. (See "Function References" in Section II and "Main Programs and Subprograms" in Section IV.)

In the following example, program EX references the function LARGE to assign a value to the variable, Z.

```
FUNCTION LARGE(A,B)

REAL A,B

IF (A .LT. B) LARGE = B

IF (A .GT. B) LARGE = A

END


PROGRAM EX

REAL X,Y
.
.
.
Z X = LARGE(X,Y)
.
.
.
END
```

BASIC EXTERNAL FUNCTIONS

Some basic computational procedures (such as taking the square root of a number) are defined in FORTRAN as basic external functions. To use these functions, the function reference, along with the appropriate actual arguments, must appear in an expression.

The type of the function (if other than Integer or Real) must be defined in the user's program through a Type statement or use of an IMPLICIT statement. When the basic external function is referenced, the actual arguments are checked for proper type. The defined function type is associated with the results.

The user can define a function subprogram with the same name as a basic external function. The new function takes the place of the system defined function.

The defined basic external functions available are shown in Table 7-2. For complete details of basic external function see *HP 3000 Compiler Library (03000-90009).*

Table 7-2. Basic External Functions

Basic External Function	Definition	Number of Arguments	Function Reference	Type of Argument	Type of Function
Exponential	e^{a}	1	EXP(a)	Real	Real
		1	DEXP(a)	Double	Double
		1	CEXP(a)	Complex	Complex
Natural logarithm	$\text{Log}_e (a)$	1	ALOG(a)	Real	Real
		1	DLOG(a)	Double	Double
		1	CLOG(a)	Complex	Complex
Common Logarithm	$\text{Log}_{10} (a)$	1	ALOG10(a)	Real	Real
		1	DLOG10(a)	Double	Double
Trigonometric Sine	Sin (a)	1	SIN(a)	Real	Real
		1	DSIN(a)	Double	Double
		1	CSIN(a)	Complex	Complex
Trigonometric cosine	Cos (a)	1	COS(a)	Real	Real
		1	DCOS(a)	Double	Double
		1	CCOS(a)	Complex	Complex
Trigonometric tangent	Tan (a)	1	TAN(a)	Real	Real
Hyperbolic tangent	Tanh (a)	1	TANH(a)	Real	Real
Square root	$(a)^{1/2}$	1	SQRT(a)	Real	Real
		1	DSQRT(a)	Double	Double
		1	CSQRT(a)	Complex	Complex
Arctangent	Arctan (a)	1	ATAN(a)	Real	Real
		1	DATAN(a)	Double	Double
	Arctan (a_1/a_2)	2	ATAN2(a_1, a_2)	Real	Real
		2	DATAN2(a_1, a_2)	Double	Double
Remaindering[2]	a_1 (mod a_2)	2	DMOD(a_1/a_2)	Double	Double
Modulus		1	CABS(a)	Complex	Real

[1]Double = double precision real.

[2]The function MOD or AMOD (a_1, a_2) is defined as $a_1 - |a_1/a_2|a_2$, where $|x|$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

SECTION VIII

Input/Output Statements

Input/output statements transfer information between data elements in memory and external devices or between data elements in memory and user-defined buffers in memory.

An input/output statement can contain a list of names of simple variables, array elements, arrays, and function subprograms. When an input statement is executed, the input values from the external device or user-buffer are assigned to the symbolic names in the list. When an output statement is executed, the values assigned to the listed variables are transferred to the external device or to the specified buffer space in memory.

To reference a specific external device, an input or output statement references the file number associated with the device. The file facility transfers data to and from the external devices. (See Section X for a discussion of the FORTRAN/3000 file facility.)

Data is input or output in groups called records. These records may be formatted or unformatted. To transfer formatted data, a string of characters known as format and edit specifications must be used to convert the data to and from memory.

All input/output is handled through the FORTRAN Formatter program. For a complete description of the format and edit specifications and the interaction between the user program and the FORTRAN Formatter, consult Section IX.

READ STATEMENTS

Read statements transfer information from an external device or character buffer in memory to specified data elements in a list. The form of a READ statement is

READ (*control part*) *element, element, ..., element*

or

READ (*control part*)

where *control part* consists of

(unit, format, labels)

or

(unit, format)

or

(unit, labels)

or

(unit)

element is a simple variable name, array or array element name, function *subprogram name or DO-implied list (see below).*

Control Part

The control part of a READ statement consists of a unit reference, format reference, or action label reference in any combination specified above.

UNIT REFERENCE. *unit* specifies the source from which the data values are to be read (either a file associated with an external device, or a user-defined buffer in core).

If the source is a file, *unit* consists of

file	*(sequential access)*
file @ record	*(direct access)*
file	is a positive integer constant or integer simple variable which indicates the desired file number (between 1 and 99, inclusive).
record	is an arithmetic expression of any type except complex, This value is converted to type integer and indicates a specific record of a direct-access file. The @ must be used to separate the file number from the record expression.

If the source is a buffer in memory, unit consists of

name

where *name* is a simple character or character array element specifying the buffer which contains the data to be transferred.

FORMAT REFERENCE. *format* specifies the location of the format and/or edit specifications which determine how the data is to be converted. *format* consists of

statement label
or
array name
or
character variable name *(Simple or subscripted)*
or
*

where *statement label* indicates that format and edit specifications appear in the FORMAT statement prefixed by statement label, and *array name* or *character variable name* indicates that format and edit specifications are contained in the array or variable specified. An asterisk (*) indicates that data is in free-field format and does not require any format or edit specifications other than those already contained within the data to be transmitted.

If *format* is omitted in the READ statement *control part*, a binary transfer takes place.

ACTION LABEL REFERENCES. *labels* allow program control over exceptional conditions and take the form

END = *statement label*

or

ERR = *statement label*

or

END = *statement label,* ERR = *statement label*

or

ERR = *statement label*, END = *statement label*

END = *statement label* transfers program control to the statement identified by *statement label* if an end-of-file condition occurs (insufficient records, record too short for a binary read, etc.). ERR = *statement label* transfers control to the statement shown if a transmission error occurs (parity error, incorrect type, etc.).

If the action labels are left out of a READ statement, and an exceptional condition occurs, the FORTRAN/3000 program is terminated with the appropriate diagnostic message. For example,

```
            PROGRAM EXRE
            INTEGER A,B
            READ (3    3,210,END = 250,ERR = 260) A,B
    210     FORMAT ...
            .
            .
            .
    250     CALL ENDOF
            .
            .
            .
    260     CALL ERROF
            .
            .
            .
```

program EXRE reads two values from a file and assigns them to the integer variables A and B. The data read is converted through a FORMAT statement (labeled 210) containing format specifications. If an end-or-file condition occurs, program control is passed to statement 250, a call to subroutine ENDOF to process the end-of-file condition. If a data transmission error occurs, control is passed to statement 260, which calls for another subroutine (ERROF) to process the error condition.

READ STATEMENT EXECUTION. When the READ statement is executed, data values are transferred from the source indicated by *unit* in the READ statement *(control part)* to the data elements specified in the element list. Elements are assigned values left-to-right as they appear in the list.

Each READ statement begins reading values from a fresh record of the file, ignoring any values left unread in records accessed by previous READ statements. If a READ statement does not contain any elements, the "next record" pointer advances one record and no transfer of data takes place.

If the READ statement contains no format reference in the *control part,* a binary read is initiated. For a sequential file *(unit = file)*, records are read sequentially until the last element in the element list receives a value. For a direct access file *(unit = file record),* only one record is read. The record must contain enough values so that all the elements in the element list receive a value. Otherwise, a run error occurs.

If the READ statement does contain a format reference in the *control part,* a formatted read is initiated. Records are read sequentially until all of the list elements have received a value, regardless of whether the file is direct or sequential access.

Array names appearing in an element list stand for all the elements of the array. Values are transferred to the array elements in the order prescribed by the array successor function (see Section IV).

ACCEPT STATEMENTS

An ACCEPT statement is a read statement intended for (but not restricted to) programs operating from a terminal device. The form is

ACCEPT *element, element, ..., element*

where *element* is a simple variable name, array name, array element, function sub-program name, or DO-implied list. (See "DO-implied Lists" in this section.)

When the ACCEPT statement is executed, it prints a question mark on the standard output device, e.g., a teleprinter. It then performs a free-field read from the standard input device (which may also be the teleprinter). For example,

```
PROGRAM RL
INTEGER A, B, C
ACCEPT A,B,C
END
```

When the program executes, it types the message ?. The user answers, for example, 35,455,733. This assigns the value 35 to A, 455 to B and 733 to C.

WRITE STATEMENTS

WRITE statements transfer information from specified data elements in memory to an external device or character buffer in core. The form of a WRITE statement is

WRITE *(control part) element, element, ..., element*

or

WRITE *(control part)*

where *control part* consists of

(unit, format, labels)

or

(unit, format)

or

(unit, labels)

or

(unit)

and *element* is a simple variable name, array name, array element, function subprogram name (defined in the same program unit only), or DO-implied list. An expression of any type can also be used. The value of the expression is transferred.

Control Part

The control part of WRITE statement consists of a unit reference, format reference or action label reference in any combination specified above.

UNIT REFERENCE. *unit* specifies the destination to which the data element values are to be transferred (either a file associated with an external device or a user-defined buffer in core).

If the destination is a file, *unit* consists of

file	*(Sequential access)*
or	
file @ record	*(Direct access)*
file	is a positive integer constant or integer simple variable which indicates the desired file number (between 1 and 99, inclusive).
record	is an arithmetic expression of any type except complex. This value is converted to type integer and indicates a specific record of a direct-access file. The @ must be used to separate the file number from the record expression.

If the destination is a buffer in memory, *unit* consists of:

name	where *name* is a simple character or character array element specifying the buffer to which the data is transferred.

FORMAT REFERENCE. *format* specifies the location of the format and/or edit specifications which determine how the data is to be converted. *format* consists of:

statement label

or

array name

or

character variable name

or

*

statement label indicates that format and edit specifications appear in the FORMAT statement prefixed by statement label. *array name* or *character vaiable name* indicates that the format and edit specifications are contained in the array or variable specified. An asterisk (*) indicates that the data is in free-field format and does not require any format or edit specifications other than those already contained within the data to be transmitted.

If format is omitted in the WRITE statement control part, a binary transfer takes place.

ACTION LABEL REFERENCES. *labels* allow program control over exceptional conditions and takes the form

END = *statement label*

or

ERR = *statement label*

or

END = *statement label,* ERR = *statement label*

or

ERR = *statement label,* END = *statement label*

END = *statement label* transfers program control to the statement identified by *statement label* if an end-of-file condition occurs (insufficient records, record too short for a binary read, etc.). ERR = *statement label* transfers control to the statement shown if a transmission error occurs (parity error, incorrect type, etc.). If the action labels are left out of a WRITE statement and an exceptional condition occurs, the FORTRAN/3000 program is terminated with the appropriate diagnostic message. (See the example in "Action Label References" for READ statements in this section.)

WRITE STATEMENT EXECUTION

When the WRITE statement is executed, values are transferred from the data elements in the *element* list to the destination indicated by *unit* in the WRITE statement control part. Values are transferred left-to-right as the elements appear in the list.

Each WRITE statement begins writing values into a fresh record of the file, ignoring any space left unused in records accessed by previous WRITE statements. If a WRITE statement does not contain any elements, the "next record" pointer advances one record and no transfer to data takes place.

If the READ statement contains no format reference in the control part, a binary read is initiated. For sequential file (*unit* = *file*) records are written sequentially until the last list element's value has been transmitted. For a direct-access file (*unit* = *file record*), only one record is written. The record size must be large enough to store all the values indicated in the *element* list of the WRITE statement. Otherwise, a run error occurs.

If the WRITE statement does contain a format reference in the control part, a formatted write is initiated. Records are written sequentially until all of the list elements' values have been transmitted, regardless of whether the file is direct-or sequential-access. An unformatted WRITE statement must have an *element* list.

Array names appearing in an *element* list stand for all the elements of the array. Values are transferred from the array elements in the order prescribed by the array successor function (see Section IV).

DISPLAY STATEMENTS

A DISPLAY statement is a write statement intended for (but not restricted to) programs operating from a terminal device. The form is

DISPLAY *element, element, ..., element*

element

where *element* is a simple variable, array name, array element or function subprogram name, or DO-implied list (see below). An expression of any type can also be used. The expression is evaluated and that value stored and transmitted as a variable.

When a DISPLAY statement is executed, the values of the data elements in the list are output in free-field format onto the teleprinter. The DISPLAY statement creates as many records, e.g., lines as needed to output all of the element values in the list.
For example,

```
PROGRAM EX
INTEGER A,B,C
A = 7
B = 45
C = 7666
DISPLAY A,B,C
END
```

When the DISPLAY statement is executed, the program types

```
7    45    7666
```

DO-implied Lists

READ, WRITE, ACCEPT, and DISPLAY statements can contain DO-implied lists. A DO-implied list contains a list of data elements to be input (read) or output (written), and a set of indexing parameters. The form of a DO-implied list is

(element, element, ..., element, var = m_1, m_2, m_3*)*

or

element, element, ..., element, var = m_1, m_2)

For a READ statement *element* is a simple variable name, array name, array element, or a function subprogram name (used as a simple variable in this case.)

For a WRITE statement

element	is a simple variable name, array name, array element, function subprogram name or an expression.
var	is an integer simple variable used as an index variable.
m_1,m_2,m_3	are all arithmetic expression of any type except complex. The are converted to type integer when used.

The DO-implied list acts as a DO loop (see Section V). The range of the implied DO loop is the list of elements to be output/input. The implied DO loop can transfer a list of simple variables or array elements, etc., or any combination of allowable data elements. The control variable is assigned the value of m_1 at the start of the loop.

The list of elements is transmitted. The control variable is then incremented by the value of m_3 or by one if m_3 is absent. The control variable is compared with m_2.

If m_3 is positive and the control variable is greater than m_2, the implied DO statement terminates; otherwise, the list is transmitted again.

If m_3 is negative and the control variable is less than m_2, the implied DO statement terminates; otherwise, the list is transmitted again. For example,

```
WRITE (IUNIT,*) (A,I = 1,3)
```

where (A,I = 1,3) is a DO-implied list. A is a simple variable. The effect of the DO-implied list is to write the value of A three times in succession. If A = 35.6, the output would consist of one record containing 35.6 35.6 35.6.

The preceding example is comparable to (but more efficient than)

```
          DO 10 I = 1,3

          WRITE (IUNIT,*) A

   10     CONTINUE
```

which results in three records, each containing 35.6:

```
35.6
35.6
35.6
```

If the *element* list of an implied DO contains several simple variables, each of the variables in the list is output/input for each pass through the loop:

```
READ (IUNIT,*) (A,B,C,J = 1,2)
```

is the same as

```
READ (IUNIT,*) A,B,C,A,B,C.
```

A DO-implied list also can transmit arrays and array elements:

```
WRITE (IUNIT,*) (A(I), I = 1,10)
```

which results in the array elements written in the order:

```
A(1) A(2) A(3) A(4) A(5) A(6) A(7) A(8) A(9) A(10)
```

If an array name is used in an *element* list, the entire array is transmitted:

```
PROGRAM DOIMP

DIMENSION A(5)

WRITE (IUNIT) A

END
```

The WRITE statement writes

```
A(1) A(2) A(3) A(4) A(5).
```

The result of the following two statements

```
DIMENSION A(3)

WRITE (IUNIT,*) (A,I = 1,2)
```

is to write the elements of array A twice:

```
A(1) A(2) A(3) A(1) A(2) A(3)
```

DO-implied lists also can be nested to transmit arrays of more than one dimension. The form of a nested DO-implied list is

$((element, ..., element, var_1 = m_1, m_2, m_3), var_2 = m_1, m_2, m_3))$

The nesting of DO-implied lists follows the same rules as nested DO loops. For example,

```
WRITE (IUNIT,*) ((A(I,J),I = 1,2), J = 1,2)
```

produces the following output:

```
A(1,1) A(2,1) A(1,2) A(2,2)
```

AUXILIARY INPUT/OUTPUT STATEMENTS

Auxiliary input/output statements are useful primarily for control of magnetic tape files. These statements are in the form:

REWIND *file*

BACKSPACE *file*

ENDFILE *file*

where *file* is a file reference consisting of a positive integer constant or integer simple variable within the value range of from 1 through 99.

The REWIND command positions the "record pointer" to the first record of the referenced file. This may envoke a physical rewind of the device associated with the file.

A BACKSPACE command positions the "record pointer" to the previous record of the file referenced.

ENDFILE writes an end-of-file record for devices which require such records, and closes the file. The file can be reopened by other I/O statements at a later time.

If a file is referenced in one of the above statements and does not have the physical capability to perform the request, the statement causes no action.

FORMAT STATEMENTS

A FORMAT statement provides the editing specifications necessary for the FORTRAN formatter to convert binary (internal) data to external string data on output, or from external string form to internal binary form on input. The form is

label FORMAT *(edit specifications)*

label	a statement label consisting of a positive integer constant from 1 to 99999
(edit specifications)	the symbols which describe the conversion of the data during transmission in (or out) of memory (see Section IX.

FORMAT statements may appear anywhere in the program.

When *label* is specified in a READ or WRITE statement control part, the formatter uses the edit specifications in the referenced FORMAT statement. Edit specifications can also be contained in a character variable or in an array. In that case, the control part of the input/output statement contains a reference which directs the Formatter to the storage space containing the *(edit specifications)*.

SECTION IX
The Formatter

The Formatter is a subroutine called by FORTRAN compiler-generated code or by SPL/3000 user programs. The FORTRAN/3000 compiler interprets READ or WRITE statements of a FORTRAN program to generate the calls to the Formatter; an SPL/3000 user must generate the calls himself. The Formatter can perform the following functions:

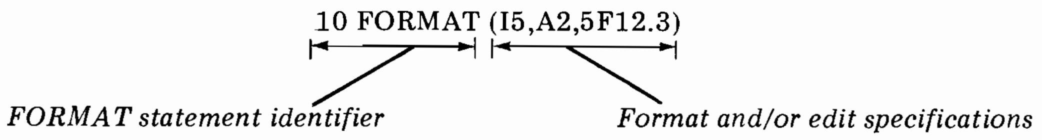
1. Convert between external ASCII numeric and/or character records and an internally represented list of variables. Formatting proceeds according to implicit parameters derived from a FORTRAN program's FORMAT statements or explicit parameters written into an SPL/3000 program.
2. Convert free-field external ASCII records to an internally represented list of variables according to format and/or edit control characters imbedded in the input records.
3. Convert an internally represented list of variables to external ASCII records which are free-field input-compatible.
4. Convert between an internally represented list of variables and a user-defined ASCII buffer storage area (core-to-core).
5. Transfer (unformatted and without conversion) between an internally represented list of variables and external files on disc or tape.

READ and WRITE statements in a FORTRAN program must meet the syntactic requirements of that language. The Formatter derives format and edit parameters from FORMAT statements or the data.

FORMAT STATEMENTS

FORMAT statements in a FORTRAN program enclose a series of format and/or edit specifications in parentheses. The specifications must be separated by commas or record terminators (see "/Edit Descriptor").

EXAMPLE:



```
10 FORMAT (I5,A2,5F12.3)
```

FORMAT statement identifier
Format and/or edit specifications

These format and edit specifications can include another set of format and/or edit specifications enclosed in parentheses; this is called nesting. The HP 3000 Formatter allows nesting to a depth of four levels.

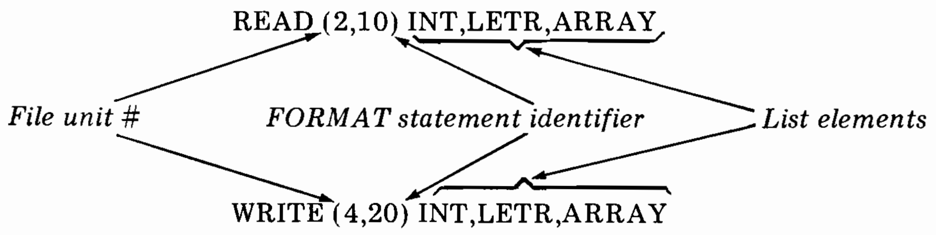
EXAMPLE:

```
20 FORMAT (I3,E12.5,3(D14.3,I6),4HSTOP)
```

READ or WRITE Statements

Formatted READ or WRITE statements in a FORTRAN program identify the list of variables that reference a FORMAT statement. (More than one READ or WRITE statement can reference a given FORMAT statement.)

EXAMPLE:



```
READ (2,10) INT,LETR,ARRAY
WRITE (4,20) INT,LETR,ARRAY
```

The list of variables can consist of any number of elements (including zero elements); there need not be a direct relationship to the number of list elements and the number of format and/or edit specifications. Refer to "Unlimited Groups," in this section.

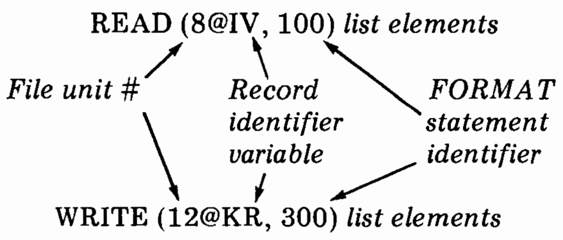
Disc Input/Output

Two types of access to files on disc devices are available through the MPE/3000 file system: sequential or direct. Either type can be established through the MPE/3000 file intrinsic FOPEN; direct access includes the capability of sequential access.

When formatted/sequential access is used, the READ or WRITE statements of a FORTRAN program are written as described above, under "READ or WRITE Statements."

When formatted/direct access is used, the READ or WRITE statements of a FORTRAN program must specify an integer, real, or double precision simple variable or a constant for the record identifier.

EXAMPLES:



```
READ (8@IV, 100) list elements
WRITE (12@KR, 300) list elements
```

When the file is opened (through the MPE/3000 file intrinsic FOPEN), the record size can be left at the system default value 128, or the user can specify a different size.

In sequential access, as many records as needed are used in sequence until the entire list of variables has been transmitted.

In direct access, only one record is transmitted.

FORMAT SPECIFICATIONS

Format specifications are written as

- A field descriptor
- A scale factor followed by a field descriptor
- A repeat specification followed by a field descriptor
- A scale factor followed by a repeat specification and a field descriptor

A brief discussion of field descriptors follows; detailed descriptions appear later in this section.

Field Descriptors

For output of data, the field descriptor determines the components of a data field into which a given list element will be written. For input, the field descriptor defines only the field width from which data can be read into an internal list element.

DECIMAL NUMERIC CONVERSIONS

Seven descriptor forms are provided:

D*w.d*	Output in double precision, floating point (with an exponent field) form.
E*w.d*	Output in real, floating point (with an exponent field) form.
F*w.d*	Output in real, fixed point (with *no* exponent field) form.
G*w.d*	Output in either the F*w.d* format or the E*w.d* format, depending on the relative size of the number to be converted.
M*w.d*	Output in monetary (business) form (real, fixed-point, plus $ and commas), e.g., $4,376.89.

N*w.d*	Output in numeration form (same as the M*w.d* format, but without the $), e.g., 3,267.54.
I*w*	Output in integer form.

where

w = the length of the external data field, in characters; must be greater than zero.

d = the number of fraction field digits in a floating or fixed point output (see detailed descriptions on the following pages). On input, *if* the external data does *not* include a decimal point, the integer is multiplied by 10^{-d}. If the external data does include a decimal point, this specification has no effect. Where listed above, *d* must be stated even if zero.

Rules for Input

All of the field descriptors listed on the preceding page accept ASCII numeric input in the following formats.

NOTE: I*w*, *on input, is interpreted as* F*w*.0

1. A series of integer number digits with or without a sign

   2314 or +56783 or –96

2. Any of the above with an exponent field with or without a sign

   2314+2 or +56783E–4 or –96D+4

3. A series of real number digits with or without a sign

   2.314 or +567.83 or –.96

4. Any of the above, with an exponent field with or without a sign

   2.314+2 or +567.83E–4 or –.96D+4

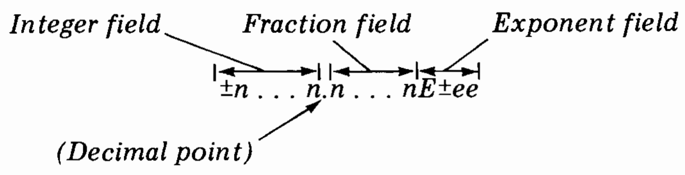
5. Either of the above items 1 and 3, in monetary (business) form

   $234 or $5,678.30 or –.96

6. Either of the above items 1 and 3, in numeration form

   2.314 or +5,678.30 or –961,534.873

In summary, the input field can include integer, fraction, and exponent subfields:



Rules: 1. The number of characters in the input field, including $ and commas, must not exceed *w* in the field descriptor used.

2. The exponent field input can be any of several forms:

+*e*	+*ee*	E*e*	E*ee*	D*e*	D*ee*
–*e*	–*ee*	E+*e*	E+*ee*	D+*e*	D+*ee*
		E–*e*	E–*ee*	D–*e*	D–*ee*

where *e* is an exponent value digit.

3. Imbedded or trailing blanks (to the right of any character read as a value) are treated as zeros; leading blanks are ignored; a field of all blanks is treated as zero.

EXAMPLES:

1Δ23 = 1023	.2Δ56ΔE+Δ4 = .20560E+04
12.Δ34 = 12.034	2Δ2,Δ45.ΔΔ3 = 202045.003
–$1,Δ34.ΔΔ5 = –1034.005	2.Δ02–Δ13 = 2.002–013

4. The type of the internal storage is independent of either the ASCII numeric input or the field descriptor used to read the input. The data is stored according to the type of the list element (variable) currently using the field descriptor. The conversion rules are as follows:

- Type INTEGER truncates a fractional input.
- Type REAL rounds a fractional input.
- Type DOUBLE PRECISION rounds a fractional input.

OCTAL NUMERIC CONVERSION

One descriptor form is provided:

O*w* for octal numbers 0 through 177777_8

where

w is the length (in characters) of the external data field (must be greater than zero).

This field descriptor accepts ASCII numeric input up to six octal digits long. Non-numeric or non-octal characters cause a conversion error.

LOGICAL CONVERSION

One descriptor form is provided:

L*w*	for logical values (T or F followed by any other characters).

The field descriptor accepts any ASCII characters input that begins with either T or F.

ALPHAMERIC CONVERSIONS

Three descriptor forms are provided:

A*w*	for alphameric conversion to and from the leftmost bytes of a list element.
R*w*	for alphameric characters to and from the rightmost bytes of a list element.
S	for alphameric characters to and from a character string (user-defined character list element).

Each of the above field descriptors accepts (but provides differing storage of) any ASCII character's input, including blanks.

D$w.d$

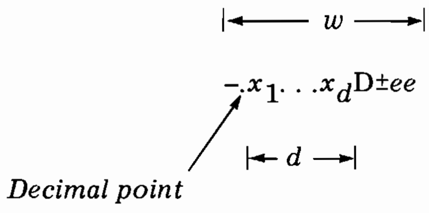
Double precision numbers

FUNCTION: Define a field for a double precision number with an exponent (floating-point).

OUTPUT

On output, the D field descriptor causes normalized output of a variable (internal representation value: interger, real, or double precision) in ASCII character floating-point form right-justified. The least significant digit of the outpt is rounded.

The external field is w positions of the record:



where

$x_1 \ldots x_d$	=	the most significant digits of the value
ee	=	the digits of the exponent value
w	=	the width of the external field
d	=	in the number of significant digits allowed in w
– (minus)		is present if the value is negative

The field width w must follow the general rule

$$w \geqslant d + 6$$

to provide positions for the sign of the value, the decimal point, d digits, the letter D, the sign of the exponent, and the exponent's two digits. If w is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If w is less than the number of positions required for the value (with the sign, decimal point, and exponent field), the entire field is filled with #'s.

EXAMPLES:

Descriptor	Internal Value	Output
D10.3	+12.342	ΔΔ.123D+02
D10.3	–12.341	Δ–.123D+02
D12.4	+12.340	ΔΔΔ.1234D+02
D12.4	–12.345	ΔΔ–.1235D+02
D7.3	+12.343	#######
D5.1	+12.344	#####

If rounding of the least significant digit occurs and "rollover" results (for example, 99.99 becomes 100.00), the rollover value is normalized and the exponent is adjusted.

EXAMPLES:

Descriptor	Internal Value	Output
D11.5	–999.997	–.10000D+04
D11.5	+999.996	Δ.10000D+04
D10.5	–99.9995	##########

INPUT

On input, the D field descriptor causes interpretation of the next *w* positions in an ASCII input record. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

All rules for input to decimal numeric conversions (see "Rules for Input") apply.

E$w.d$

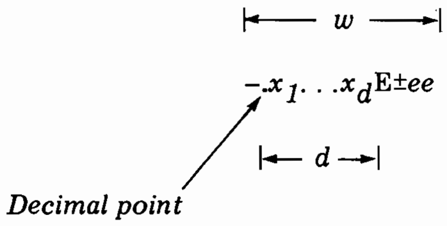
Real Numbers

FUNCTION: Define a field for a real number with an exponent (floating-point).

OUTPUT

On output, the E field descriptor causes normalized output of a variable (internal representation value: integer, real, or double precision) in ASCII character floating-point form, right-justified. The least significant digit of the output is rounded.

The external field width is w positions in the record:



where

$x_1..x_d$	=	the most significant digits of the value
ee	=	the digits of the exponent value
w	=	the width of the external field
d	=	the number of significant digits allowed in w (for output, d must be greater than zero
–(minus)		is present if the value is negative

The field width w must follow the general rule

$$w \geqslant d + 6$$

to provide positions for the sign of the value, the decimal point, d digits, the letter E, the sign of the exponent, and the exponent's two digits. If w is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If w is less than the number of positions required for the value (with the sign, decimal point, and exponent field), the entire field is filled with #'s.

EXAMPLES:

Descriptor	Internal Value	Output
E10.3	+12.342	ΔΔ.123E+02
E10.3	–12.341	Δ–.123E+02
E12.4	+12.340	ΔΔΔ.1234E+02
E12.4	–12.345	ΔΔ–.1235E+02
E7.3	+12.34	#######
E5.1	+12.34	#####

If rounding of the least significant digit occurs and "rollover" results (for example, 99.99 becomes 100.00), the rollover value is normalized and the exponent is adjusted.

EXAMPLES:

Descriptor	Internal Value	Output
E11.5	–999.998	–.10000E+04
E11.5	999.995	Δ.10000E+04
E10.5	–99.9997	##########

INPUT

On input, the E field descriptor causes interpretation of the next *w* positions in an ASCII input record. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

All rules for input to decimal numeric conversions (see "Rules for Input") apply.

F*w.d*

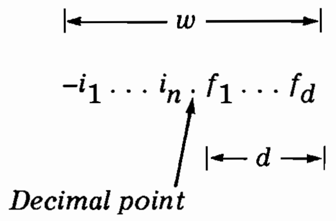
Real Numbers

FUNCTION: Define a field for a real number without an exponent (fixed-point).

OUTPUT

On output, the F field descriptor causes output of a variable (internal representation value: integer, real, or double precision) in ASCII character fixed-point form, right-justified. The least significant digit of the output is rounded.

The external field width is w positions in the record:



where

$i_1 \ldots i_n$	=	the integer digits
$f_1 \ldots f_d$	=	the fraction digits
w	=	the width of the external field
d	=	the number of fractional digits allowed in w
n	=	the number of integer digits
– (minus)		is present if the value is negative.

The field width w must follow the general rule

$$w \geqslant d + n + 3$$

to provide positions for the sign, n digits, the decimal point, d digits, and a rollover digit if needed (see the following examples). If w is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If w is less than the number of positions required for the value (with the sign and decimal point), the entire field is filled with #s.

EXAMPLES:

Descriptor	Internal Value	Output
F10.3	+12.3402	ΔΔΔΔ12.340
F10.3	–12.3413	ΔΔΔ–12.341
F12.3	+12.3434	ΔΔΔΔΔΔ12.343
F12.3	–12.3456	ΔΔΔΔΔ–12.346
F4.3	+12.34	####
F4.3	+12345.12	####

If rounding of the least significant digit occurs and "rollover" results (for example, 99.99 becomes 100.00), the stated formula for *w* provides enough positions for the value.

EXAMPLES:

Descriptor	Internal Value	Output
F8.2	+999.997	Δ1000.00
F8.2	–999.996	–1000.00
F7.2	–999.995	#######

INPUT

On input, the F field descriptor causes interpretation of the next *w* positions in an ASCII input record. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

All rules for input to decimal numeric conversions (see "Rules for Input") apply.

G*w.d*

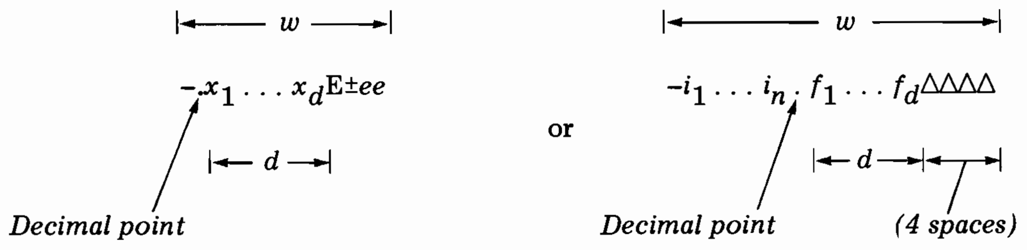
Real Numbers

FUNCTION: Define a field for a real number without an exponent (fixed-point) or, if needed, with an exponent (floating-point).

OUTPUT

On output, the G field descriptor causes output of a variable (internal representation value: integer, real, or double precision) in ASCII character fixed-point form, or if needed, floating-point form, right-justified. The least significant digit of the output is rounded.

The external field is w positions in the record:



where

$i_1 \ldots i_n$	= the integer digits	(F*w.d* descriptor)
$f_1 \ldots f_d$	= the fraction digits	(F*w.d* descriptor)
$x_1 \ldots x_d$	= the most significant digits of the value (E*w.d* descriptor)	
ee	= the digits of the exponent value (E*w.d* descriptor)	
w	= the width of the external field	
d	= the number of fractional digits allowed in w	
n	= the number of integer digits (F*w.d* descriptor)	
– (minus)	is present if the value is negative	

The G*w.d* field descriptor is interpreted as an F*w.d* descriptor for fixed-field form or as an E*w.d* descriptor for floating-point form, according to the internal representation absolute value (N) after rounding. If the number of integer digits in N is $> d$, or if N $<$ .1, the E descriptor is used; otherwise the F descriptor is used (see following page).

IF		$N < 0.1$	THEN E*w*.*d*;
IF	0.1	$\leqslant N < 1$	THEN F(*w*–4) .*d* plus 4X (spaces);
IF	1	$\leqslant N < 10^1$	THEN F(*w*–4) .(*d*–1) plus 4X;
IF	10^1	$\leqslant N < 10^2$	THEN F(*w*–4) .(*d*–2) plus 4X;
IF	10^2	$\leqslant N < 10^3$	THEN F(*w*–4) . (*d*–3) plus 4X;
⋮	⋮	⋮	⋮
IF	$10^{(d-1)}$	$\leqslant N < 10^d$	THEN F(*w*–4) .0 plus 4X;
IF	10^d	$\leqslant N$	THEN E*w*.*d*;

EXAMPLES:

G12.6, N = 1234.5: F(*w*–4) . (*d*–4) = F8.2, 4X: △1234.50△△△△

G13.7, N = 123456.7: F(*w*–4) . (*d*–6) = F9.1, 4X: △123456.7△△△△

G9.2, N = 123.4: E*w*.*d* = E9.2: △△.12E+03

The field width *w* must follow the general rule for the E*w*.*d* descriptor

$$w \geqslant d + 6$$

to provide positions for the sign of the value, *d* digits, the decimal point (preceding x_1), and, if needed, the letter E, the sign of the exponent, and the exponent's two digits. If *w* is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If *w* is less than the number of positions required for the value (with the sign, decimal point, and the exponent field—or 4 spaces), the entire field is filled with #'s.

EXAMPLES:

Descriptor	Internal Value	Output
G10.3 (E10.3)	+1234	△△.123E+04
G10.3 (E10.3)	–1234	△–.123E+04
G12.4 (E12.4)	+12345	△△△.1235E+05
G12.4 (F8.0,4X)	+9999	△△△9999.△△△△
G12.4 (F8.1,4X)	–999	△△–999.0△△△△
G7.1 (E7.1)	+.09	△.9E–01
G5.1 (E5.1)	–.09	#####

When the E descriptor is used, if rounding of the least significant digit occurs and "rollover" results (for example, 99.99 becomes 100.00), the rollover value is normalized and the exponent is adjusted.

EXAMPLES:

Descriptor	Internal Value	Output
G12.1 (E12.2)	+9999	ΔΔΔΔΔ.10E+05
G8.2 (E8.2)	+999	Δ.10E+04
G7.2 (E7.2)	−999	######

INPUT

On input, the G field descriptor causes interpretation of the next *w* positions in an ASCII input record. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

All rules for input to decimal numeric conversions (see "Rules for Input") apply.

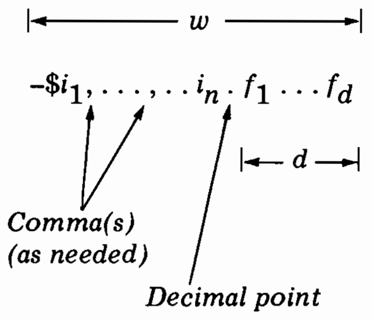
M$w.d$
Real Numbers

FUNCTION: Define a field for a real number without an exponent (fixed-point) written in monetary (business) form.

OUTPUT

On output, the M field descriptor causes output of a variable (internal representation value: integer, real, or double precision) in ASCII character fixed-point form right-justified, with a dollar sign $ and commas. The least significant digit of the output is rounded.

The external field is w positions in the record:



where

$i_1 \ldots\ldots i_n$	=	the integer digits (without commas)
$f_1 \ldots f_d$	=	the fraction digits
commas = c	=	the number of output commas needed: one to the left of every third digit left of the decimal point; see general rule for w below.
d	=	the number of fractional digits allowed in w
n	=	the number of integer digits
w	=	the width of the external field
– (minus)		is present if the value is negative

The field width w must follow the general rule

$$w \geqslant d + n + c + 4$$

to provide positions for the sign, \$, n digits, c commas, the decimal point, d digits, and a rollover digit if needed (see the following examples). If w is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left.

If *w* is less than the number of positions required for the output value (with the sign $, comma(s), and the decimal point), the entire field is filled with #'s.

EXAMPLES:

Descriptor	Internal Value	Output
M10.3	+12.3402	ΔΔΔ$12.340
M10.3	-12.3404	ΔΔ-$12.340
M13.3	+80175.3965	ΔΔ$80,175.397
M12.2	-80175.396	Δ-$80,175.40
M12.2	+28705352.563	############

If rounding of the least significant digit occurs and "rollover" results (for example, 99.99 becomes 100.00), the stated formula for *w* provides enough positions.

EXAMPLES:

Descriptor	Internal Value	Output
M12.2	+99999.996	Δ$100,000.00
M12.2	-99999.998	-$100,000.00
M11.2	-99999.995	###########

INPUT

On input, the M field descriptor causes interpretation of the next *w* positions in an ASCII input record. The field width is expected (but not required) to have a $ and comma(s) imbedded in the data as described above for M*w.d* outputs; the $ and comma(s) are ignored. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

All rules for input to decimal numeric conversions (see "Rules for Input") apply.

N$w.d$

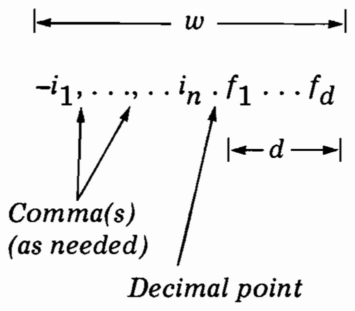
Real Numbers

FUNCTION: Define a field for a real number without exponent (fixed-point) written in numeration form (same as M$w.d$ but without \$ on output).

OUTPUT

On output, the N field descriptor causes output of a variable (internal representation value: integer, real, or double precision) in ASCII character fiexed-point form, right-justified, with commas. The least significant digit of the output is rounded.

The external field is w positions in the record:



where

$i_1 \ldots \ldots i_n$	=	the integer digits (without commas)
$f_1 \ldots f_d$	=	the fraction digits
commas = c	=	the number of output commas needed: one to the left of every third digit left of the decimal point; see general rule for w below.
d	=	the number of fractional digits allowed in w
n	=	the number of integer digits
w	=	the width of the external field
– (minus)		is present if the value is negative

The field width w must follow the general rule

$$w \geqslant d + n + c + 3$$

to provide positions for the sign, n digits, c commas, the decimal point, d digits, and a rollover digit if needed (see the following examples). If w is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If w is less than the number of positions required for the output value (with the sign, comma(s), and the decimal point), the entire field is filled with #'s.

EXAMPLES:

Descriptor	Internal Value	Output
N9.3	+12.3402	ΔΔΔ12.340
N9.3	–12.3404	ΔΔ–12.340
N12.3	+80175.3965	ΔΔ80,175.397
N11.2	–80175.396	Δ–80,175.40
N11.2	+28705352.563	###########

If rounding of the least significant digit occurs and "rollover" results (for example, 99.99 becomes 100.00), the stated formula for *w* provides enough positions.

EXAMPLES:

Descriptor	Internal Value	Output
N11.2	+99999.995	Δ100,000.00
N11.2	–99999.997	–100,000.00
N10.2	–99999.999	##########

INPUT

On input, the N field descriptor causes interpretation of the next *w* positions in an ASCII input record as a real number without exponent (fixed-point). The field width is expected (but not required) to have comma(s) imbedded in the data as described above for N*w.d* outputs; the comma(s) are ignored. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

All rules for input to decimal numeric conversions (see "Rules for Input.") apply.

I*w*

Integer Numbers

FUNCTION: Define a field for an integer number.

OUTPUT

On output, the I field descriptor causes output of a variable (internal representation value: integer, real, or double precision) in ASCII character integer form, right-justified. If the internal representation is real or double precision, the least significant digit of the output is rounded.

The external field is *w* positions of the record:

$$\vert\longleftarrow w \longrightarrow\vert$$

$$-i_1 \ldots i_n$$

where

$i_1 \ldots i_n$	=	the integer digits
n	=	the number of significant digits
w	=	the width of the external field
– (minus)		is present if the value is negative

The field width *w* must follow the general rule

$$w \geqslant n + 2$$

to provide positions for the sign, *n* digits, and a rollover digit if needed (see the following examples). If *w* is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If *w* is less than the number of positions required for the output (all digits of the integer and, when needed, the sign), the entire field is filled with #'s.

EXAMPLES:

Descriptor	Internal Value	Output
I5	–123	Δ–123
I5	+123	ΔΔ123
I5	+12345	12345
I5	–12345	#####
I4	+12.4	ΔΔ12
I4	–12.7	Δ–13
I6	–.3765E+03	ΔΔ–377

If rounding of the least significant digit occurs and "rollover" results (for example, 99.99 becomes 100.00), the stated formula for *w* provides enough positions:

EXAMPLES:

Descriptor	Internal Value	Output
I5	–999.8	–1000
I5	+999.6	Δ1000
I4	–999.5	####

INPUT

On input, the I field descriptor functions as an F*w.d* descriptor with *d* = 0; it causes interpretation of the next *w* positions in the ASCII input record. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

All rules for input to decimal numeric conversions (see "Rules for Input") apply.

Ow

Octal Integer Number

FUNCTION: Define a field for an octal integer number.

OUTPUT

On output, the O field descriptor causes output of a variable (internal representation value: integer only) in ASCII-character octal integer form, right-justified.

The external field is w positions of the record:

$$\begin{array}{c} |\leftarrow w \rightarrow| \\ i_1 \ldots i_n \end{array}$$

where

$i_1 \ldots i_n$ = the octal integer digits

n = the number of significant digits (maximum: 6)

w = the width of the external field

The field width w can be any desired value but should be $\geqslant 6$ for complete accuracy. If w is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If w is less than the number of positions required for the entire octal integer, only the w least significant digits are output.

EXAMPLES:

Descriptor	Internal Value	Output
O7	143567	Δ143567
O8	102077	ΔΔ102077
O4	027033	7033
O6	002004	002004

INPUT

On input, the O field descriptor causes interpretation of the next *w* positions in the ASCII input record as an octal integer number. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

The input field can consist of only octal digits, no more than six digits (no larger than 177777_8) are interpreted. Any non-octal or non-numeric character (including a blank) anywhere in the field will produce a conversion error. If *w* is less than 6, *w* digits are right-justified in the internal representation (one word of memory).

EXAMPLES:

Descriptor	Input	Result
O6	134577	134577
O4	275674	002756
O10	1345367421	167421

L*w*

Logical (Boolean) Values

FUNCTION: Define a field for a logical value.

OUTPUT

On output, the L field descriptor causes output of a variable (internal representation value: integer or logical (boolean)) in ASCII-character logical value form (T or F).

The external field is *w* positions of the record:

$$\vdash\!\longleftarrow w \longrightarrow\!\dashv$$

$$X_1 \ldots X_n c$$

where

$X_1 \ldots X_n$ = w–1 blanks

c = either of two logical characters: T (true) or F (false)

n = the number of blank spaces to the left of c

w = the width of the external field

The field width w can be any value $\geqslant 1$.

The logical character c is T if the least significant bit of the internal representation is 1; c is F if that bit is 0.

EXAMPLES:

Descriptor	Internal Value	Output
L1	102033_8	T
L13	$32767(77777_8)$	ΔΔΔΔΔΔΔΔΔΔΔΔT
L5	$+124(174_8)$	ΔΔΔΔF

INPUT

On input, the L field descriptor causes a scan of the next *w* positions in an ASCII input record to find a logical character (T or F). All positions to the left of the logical character must be blank; any other character(s) can follow the logical character. The character T is converted to –1 (177777_8), F is converted to 0 (000000_8).

EXAMPLES:

Descriptor	Input	Result
L8	ΔΔΔΔTRUE	177777_8
L1	F	000000_8
L6	ΔFALSE	000000_8

Aw

Leftmost ASCII Characters

FUNCTION: Define a field for ASCII alphameric characters of a variable.

OUTPUT

On output, the A field descriptor causes output of one or more bytes of a variable in ASCII-character alphameric form. The maximum number n of bytes (thus, the maximum number of characters available to a single Aw descriptor) depends on the type of the variable: for logical or integer, $n = 2$; for real, $n = 4$; for double precision, $n = 6$; for character, n = the length attribute[1] of the character variable (any integer in the range [1,255]).

The external field is w positions of the record:

$$|\xleftarrow{\quad} w \xrightarrow{\quad}|$$

$$s_1 \ldots s_r c_1 \ldots c_n$$

where

$c_1 \ldots c_n$	=	the alphameric characters
n	=	the number of characters
w	=	the width of the external field
r	=	any remaining positions not used by n ($r = w-n$)
$s_1 \ldots s_r$	=	blank spaces (when needed)

The field width w can be any value $\geqslant 1$. If w is $\geqslant n$, the output is right-justified in the field with $w-n$ blanks to the left. If w is $< n$, the leftmost w bytes of the variable are output. The $n-w$ remaining bytes are ignored.

EXAMPLES:

Descriptor	Internal Characters	Variable Type (n =)	Output
A3	SA	Logical or Integer (2)	ΔSA
A3	SAMB	Real (4)	SAM
A7	JANETW	Double Precision (6)	ΔJANETW
A10	BG	Logical or Integer (2)	ΔΔΔΔΔΔΔΔBG
A4	DIXMCG	Double Precision (6)	DIXM
A12	LEFTMOST	Character[1] (8)	ΔΔΔΔLEFTMOST
A6	LEFTMOST	Character[1] (8)	LEFTMO

[1]As defined in a Type statement such as CHARACTER*8 LOCALE.

INPUT

On input, the A field descriptor causes transmittal of w positions in an ASCII input record to n bytes of the variable (list element) currently using the field descriptor. If $w \geq n$, the first $w-n$ characters of input are skipped, and n characters are transmitted. If $w < n$, w characters are transmitted to the leftmost bytes of the variable, and all remaining $n-w$ bytes are set to blank.

EXAMPLES:

Descriptor	External Characters	Variable Type (n =)	Internal Result
A3	CAB	Integer or Logical (2)	AB
A2	CA	Integer or Logical (2)	CA
A10	COMPLEMENT	Integer or Logical (2)	NT
A4	REAL	Double Precision (6)	REALΔΔ
A4	REAL	Real (4)	REAL
A7	PROGRAM	Character[1] (8)	PROGRAMΔ

[1]As defined in a Type statement such as CHARACTER*8 LOCALE.

Rw

Rightmost ASCII Characters

FUNCTION: Define a field for ASCII alphameric characters of a variable.

OUTPUT

On output, the R field descriptor causes output of one or more bytes of a variable in ASCII character alphameric form. The maximum number n of bytes (thus, the maximum number of characters) available to a single Rw descriptor depends on the type of the variable: for logical or integer, $n = 2$; for real, $n = 4$; for double precision, $n = 6$; for character, n = the length attribute[1] of the character variable (any integer in the range [1,255]).

The external field is w positions of the record:

$$\overbrace{s_1 \ldots s_r c_1 \ldots c_n}^{w}$$

where

$c_1 \ldots c_n$	=	the alphameric characters
n	=	the number of characters
w	=	the width of the external field
r	=	any remaining positions not used by n ($r = w-n$)
$s_1 \ldots s_r$	=	blank spaces (when needed)

The field width w can be any value $\geqslant 1$. If w is $\geqslant n$, the output is right-justified in the field with $w-n$ blanks to the left. If w is $< n$, the rightmost bytes of the variable are output. The $n-w$ remaining bytes are ignored.

EXAMPLES:

Descriptor	Internal Characters	Variable Type (n =)	Output
R3	SA	Logical or Integer (2)	ΔSA
R3	SAMB	Real (4)	AMB
R7	JANETG	Double Precision (6)	ΔJANETG
R10	BG	Logical or Integer (2)	ΔΔΔΔΔΔΔΔBG
R4	DIXMCG	Double Precision (6)	XMCG
R12	RIGHTMOST	Character[1] (9)	ΔΔΔRIGHTMOST
R6	RIGHTMOST	Character[1] (9)	HTMOST

[1]As defined in a Type statement such as CHARACTER*9 LOCALE.

INPUT

On input, the R field descriptor causes transmittal of w positions in an ASCII input record to n bytes of the variable currently using the field descriptor. If $w \geqslant n$, the first $w-n$ characters of input are skipped, and n characters are transmitted. If $w < n$, w characters are transmitted to the rightmost bytes of the variable, and all bits of the remaining $n-w$ bytes are set to 0 (ASCII Null).

EXAMPLE:

Descriptor	External Characters	Variable Type (n =)	Internal Result
R3	CAB	Integer or Logical (2)	AB
R2	CA	Integer or Logical (2)	CA
R10	COMPLEMENT	Integer or Logical (2)	NT
R4	REAL	Double Precision (6)	*aa* REAL[1]
R4	REAL	Real (4)	REAL
R7	PROGRAM	Character[2] (8)	*a*PROGRAM[1]

[1]*a* = ASCII Null.
[2]As defined in a Type statement such as CHARACTER*8 LOCALE.

S

Strings of ASCII Characters

FUNCTION: Define a field for a string of ASCII alphameric characters.

OUTPUT

On output, the S field descriptor causes output of a variable[1] (internal value: character only) in ASCII-character alphameric form.

The external field is *l* positions of the record:

$$\overset{\longleftarrow\ l\ \longrightarrow}{c_1 \ldots c_n}$$

where

$c_1 \ldots c_n$	=	the alphameric characters
n	=	the number of characters
l	=	the length attribute of the character variable (list element); thus, the width of the external field

EXAMPLES:

NAME Internal Characters	Output
JIM	MY NAME IS JIM JONES
SAM	MY NAME IS SAM JONES

where the list element and length attributed are defined by the Type statement CHARACTER*3 NAME and the format and edit specifications are

("MY NAME IS ",S," JONES")

[1] If the variable (list element) is not type character,[2] SOFTERROR' message FMT: STRING MISMATCH occurs.

INPUT

On input, the S field descriptor causes transmittal of *l* positions in an ASCII input record to the character variable currently using the field descriptor.

EXAMPLES:

External Characters	DAY Internal Result
MONDAY Δ	MONDAY Δ
SATURDAY	SATURDA

where the list element and length attribute are defined by the Type statement CHARACTER*7 DAY and the format and edit specifications are

("TODAY IS ", S).

Scale Factor

The scale factor is a format specification to modify the normalized *output* of the D*w.d*, E*w.d*, and the G*w.d*-selected E*w.d*[1] field descriptors and the fixed-point *output* of the F*w.d*, M*w.d*, and N*w.d* field descriptors. It also modifies the fixed-point and integer (no exponent field) *inputs* to the D*w.d*, E*w.d*, F*w.d*, G*w.d*, M*w.d*, and N*w.d* field descriptors. The scale factor has no effect on output of the G*w.d*-selected F*w.d*[1] field descriptor or floating-point (with exponent field) inputs.

A scale factor is written in one of two forms:

*n*P*f*

or

*n*P*rf*

where

n	=	an integer constant or – (minus) followed by an integer constant: the scale value
P	=	the scale factor identifier
f	=	the field descriptor
r	=	a repeat specification—for a field descriptor (described later in this section)

When the Formatter begins to interpret a FORMAT statement, the scale factor is set to zero. Each time a scale factor specification is encountered in that FORMAT statement, a new value is set. This scale value remains in effect for all subsequent affected field descriptors or until use of that FORMAT statement ends.

EXAMPLES:

Format Specifications	Comments
(E10.3,F12.4,I9)	No scale factor change, previous value remains in effect.
(E10.3,2PF12.4,I9)	Scale factor for E10.3 unchanged from previous value, changes to 2 for F12.3, has no effect on I9.

If the FORMAT statement includes one or more nested groups (see "Nesting," this section), the last scale factor value encountered remains in effect.

[1] See descriptions for G*w.d*.

EXAMPLE:

Format Specifications	Comments
(G9.2,2PF9.4,E7.1, 2(D10.2,–1PG8.1))	Scale values resulting are

Descriptor	Scale Value
G9.2	(Unchanged from previous value)
F9.4	2
E7.1	2
D10.2	2
G8.1	–1
D10.2	–1
G8.1	–1

OUTPUT

On output, the scale factor affects D*w*.*d*, E*w*.*d*, F*w*.*d*, M*w*.*d*, N*w*.*d*, and G*w*.*d*-selected E*w*.*d* field descriptors only.

D*w*.*d* and E*w*.*d*

The internal fraction is multiplied by 10^n, and the internal exponent value is reduced by n.

- If $n \leqslant 0$, the output fraction field has $-n$ leading zeros, followed by $d + n$ significant digits. The least significant digit is rounded.
- If $n > 0$, the output has n significant digits in the integer field, and $(d - n) + 1$ digits in the fraction field. The least significant digit field is rounded.
- The field width specification w normally required may have to be increased by 1.

EXAMPLES:

Scale Factor[1] and Field Descriptor	Internal Value	Output
E12.4	+12.345678	ΔΔΔ.1235E+02
3PE12.4	+12.345678	ΔΔ123.46E–01
–3PE12.4	+12.345678	ΔΔΔ.0001E+05

[1] In "Examples," no scale factor stated implies zero.

F*w.d*, M*w.d*, and N*w.d*

The internal value is multiplied by 10^n, then output in the normal manner.

EXAMPLES:

Scale Factor[1] and Field Descriptor	Internal Value	Output
F11.3	1234.500	ΔΔΔ1234.500
–2PF11.3	1234.500678	ΔΔΔΔΔ12.345
2PF11.3	1234.500678	Δ123450.068
1PM11.3	1234.500678	$12,345.007

G*w.d*-selected E*w.d*

The effect is exactly as described for E*w.d*.

G*w.d*-selected F*w.d*

The scale factor has no effect.

INPUT

On input, the scale factor effect is the same for integer or fixed-field (no exponent field) inputs to the D*w.d*, E*w.d*, F*w.d*, G*w.d*, M*w.d*, and N*w.d* field descriptors. The external value is multiplied by 10^{-n}, then converted in the usual manner.

If the input includes an exponent field, the scale factor has no effect.

EXAMPLES:

Scale Factor[1] and Field Descriptor	External Value	Internal Representation
E10.4	123.9678	.1239678E+03
2PD10.4	123.9678	.1239678E+01
–2PG11.5	123.96785	.12396785E+05
–2PE13.5	1239.6785E+02	.12396785E+06

[1] In "Examples," no scale factor stated implies zero.

Repeat Specification—For Field Descriptors

The repeat specification is a positive integer written to the left of the field descriptor it controls. If a scale factor is also needed, it is written to the left of the repeat specification.

The repeat specification allows one field descriptor to be used for several list elements. It can also be used for nested (groups of) format specifications.

EXAMPLES:

```
(4E12.4) = (E12.4,E12.4,E12.4,E12.4)

(-2P3D8.2,2I6) = (-2PD8.2,D8.2,D8.2,I6,I6)

(E8.2/3F7.1,3(I6,4HLOAD,D12.3))
   = (E8.2/F7.1,F7.1,F7.1,I6,4HLOAD,D12.3,I6,4HLOAD,D12.3,I6,4HLOAD,D12.3)

(2(M8.2)) = (M8.2,M8.2)
```

EDIT SPECIFICATIONS

Edit specifications are written as an edit descriptor or a repeat specification followed by an edit descriptor.

NOTE: The repeat specification cannot be used directly on the nH or nX edit descriptors. See "Repeat Specification—For Edit Descriptors."

Edit Descriptors

There are six edit descriptors:

Descriptor	Function
" . . . "	Fix the next *n* characters of an edit specification.
' . . . '	Fix the next *n* characters of an edit specification.
*n*H	Initialize the next *n* characters of an edit specification.
*n*X	Skip *n* positions of the external record.
T*n*	Select the position in an external record where data input/output is to begin or resume.
/	Signal the end of a current record and the beginning of a new record.

Detailed descriptions of each edit descriptor follow.

" . . . "
ASCII String (Fixed)

FUNCTION: Fix *n* characters in the edit specification where *n* is the number of ASCII characters enclosed in the *quotation marks.* Any one or more of those characters can be a quotation mark if signaled by an adjacent quotation mark. Any other ASCII characters, including ' (apostrophe), can be used without restriction.

OUTPUT

On output, the " . . . " edit descriptor causes *n* characters to be transmitted to the external record; any adjacent pair of quotation marks is transmitted as one quotation mark.

EXAMPLES:

Edit Descriptor	Output
"OUTPUT∆" "LOAD" "."	OUTPUT∆"LOAD".
"USER'S∆PROGRAM"	USER'S∆PROGRAM

INPUT

On input, the " . . . " edit descriptor causes *n* positions of the input record to be skipped. Each pair of adjacent quotation marks counts as one position.

EXAMPLES:

Edit Descriptor	Input	Comment
"HEADING∆HERE"	THIS∆IS∆THE∆START	12 positions of the input are skipped.
"HEADING∆" "A" "∆."	THIS∆IS∆THE∆END∆OF	13 positions of the input are skipped.

' . . . '
ASCII String (Fixed)

FUNCTION: Fix *n* characters in the edit specification, where *n* is the number of ASCII characters enclosed in the *apostrophes*. Any one or more of those characters can be an apostrophe if signaled by an adjacent apostrophe. Any other ASCII characters, including " (quotation mark), can be used without restriction.

OUTPUT

On output, the ' . . . ' edit descriptor causes *n* characters to be transmitted to the external record; any adjacent pair of apostrophes is transmitted as an apostrophe.

EXAMPLES:

Edit Descriptor	Output
'PRINT△' 'DATA' '.'	PRINT△ 'DATA'.
'SAM' 'S△"SCORE" '	SAM'S△ "SCORE"

INPUT

On input, the ' . . . ' edit descriptor causes *n* positions of the input record to be skipped. Each pair of adjacent apostrophes counts as one position.

EXAMPLES:

Edit Descriptor	Input	Comment
'COLUMN△HEAD'	BEGIN△DATA△INPUT	11 positions of the input are skipped
'ROW△LABEL△' 'B' '.'	END△DATA△INPUT	14 positions of the input are skipped.

*n*H

ASCII String (Variable)

FUNCTION: Initialize the next *n* characters of the edit specification. Any ASCII character is legal. If written, *n* must be a positive integer greater than zero (if omitted, its default value is 1).

OUTPUT

On output, the *n*H edit descriptor causes the current next *n* characters in the edit specification to be transmitted to the external record.

If the edit descriptor has *not* been referenced by a READ statement (see "Input"), the ASCII characters originally written into the edit descriptor are transmitted.

If the edit descriptor has been referenced by a READ statement, the ASCII characters read last are transmitted.

EXAMPLES:

Edit Descriptor	Input Last Read	Output
4HMULT	(None)	MULT
7HFORTRAN	ALGOL△△	ALGOL△△
12HPROGRAM△DATA	BINARY△LOADER	BINARY△LOADE
10HCALCULATED	PASSED△△△△	PASSED△△△△

INPUT

On input, the *n*H edit descriptor causes the next *n* characters of the external record to be transmitted to replace the next *n* characters in the edit specification.

*n*X
ASCII Blanks

FUNCTION: Skip *n* positions of the external record. If written, *n* must be a positive integer greater than zero; if omitted, the default value is 1.

OUTPUT

On output, the *n*X edit descriptor causes *n* positions of the external record to be skipped, typically to separate fields of data.

EXAMPLES:

Format/Edit Specifications	Contents of Numeric List Element(s)		Output
(E7.1,4X,"END")	34.1	*Fields:*	$\underbrace{\Delta\text{.3E+02}}_{7}\underbrace{\Delta\Delta\Delta\Delta}_{4}\text{END}$
(F8.2,2X,I6)	5.87,436	*Fields:*	$\underbrace{\Delta\Delta\Delta\Delta\text{5.87}}_{8}\underbrace{\Delta\Delta}_{2}\underbrace{\Delta\Delta\Delta\text{436}}_{6}$

NOTE: This descriptor, when used with the Tn edit descriptor (described later in this section), may cause previous characters to be overlaid.

EXAMPLE:

Format/Edit Specifications	Output
("ABCDEFG", T1, "X", 2X, "Y")	XBCYEFG

INPUT

On input, the *n*X edit descriptor causes the next *n* positions of the input record to be skipped.

EXAMPLES:

Format/Edit Specifications	External Record Input	Data Transmitted to List Elements
(D8.2,3X,M9.2)	Δ.25E+02END$1,563.79	.25E+02, 1563.79
(5X,E9.2,I5)	54321–98.7563814581	–.9876538E+02, 14581

T*n*

Position (Tabulate) Data

FUNCTION: Select the position (tabulation) in an external record where data input/output is to begin or resume.

The T*n* edit descriptor positions the record pointer to the *n*th position in the record.

OUTPUT EXAMPLES

1. **Format/Edit Specifications**

   (T10,"DESCRIPTION", T25, "QUANTITY", T1, "PARTΔNO.")

   Result

   ```
   PARTΔNO.ΔDESCRIPTIONΔΔΔΔQUANTITY
   ↖        ↖               ↖
   position #1  position #10    position #25
   ```

2. **Format/Edit Specifications**

   (T25,I3,T1,3A2,T10,3A4)

   Contents of List Elements

   125,HR124A,LOCK-WASHERS

   Result

   ```
   HR124AΔΔΔLOCK-WASHERSΔΔΔ125
   ↖        ↖               ↖
   position #1  position #10  position #25
   ```

INPUT EXAMPLE

Format/Edit Specifications

(T13,E8.2,T1,I4,T24,M12.3)

Input

```
1325COUNTEDΔΔΔ525.78LBSΔΔ$4,365.78ΔCOST
↖           ↖          ↖
position #1   position #13   position #24
```

Results in List Elements

.52578E+03, 1325, .436578E+04

As can be seen in the above examples, the position numbers *n* need not be given in ascending order.

NOTE: This descriptor may cause previous characters to be overlaid (see nX descriptions, earlier in this section).

/
Record Terminator

FUNCTION: Terminate the current external record and begin a new record (on a line printer or a keyboard terminal, a new line; on a card device, a new card; etc.).

OUTPUT and INPUT

The / edit descriptor has the same result for both output and input: it terminates the current record and begins a new record.

If a series of two or more / edit descriptors are written into a FORMAT statement, the effect is to skip *n*–1 records, where *n* is the number of /'s in the series. A series of /'s can be written by using the repeat specification.

NOTE: If one or more / edit descriptors are the first item(s) in a series of format specifications, n (not n–1) records are skipped for that series of /'s.

EXAMPLES:

Format Specifications	Output	Record #
(E12.5,I3/"END")	ΔΔ.32456E+04Δ95	1
	END	2
(E12.5,I3///"END")	ΔΔ.32456E+04Δ96	1
		2
		3
	END	4
(I5,3HEND,4/"NEW DATA")	43592END	1
		2
		3
		4
	NEW DATA	5
(2/"END")		1
		2
	END	3

The / edit descriptor can also be used without a comma to separate it from other format and/or edit specifications; it has the same separating effect as a comma.

Repeat Specification—For Edit Descriptors

The repeat specification is a positive integer written to the left of the edit descriptor it controls. It is written as *r*" . . . " or *r*' . . . ' or *r*(*n*H) or *r*(*n*X) or *r*/, where *r* is the repetition value.

> *NOTE: The forms r(nH) and r(nX) may include other field and/or edit descriptors within the parentheses.*

EXAMPLES:

```
(E9.2/3F7.1,2(4HDATA)) = (E9.2/F7.1,F7.1,F7.1,4HDATA,4HDATA)
(2(5HABORT2/)) = (5HABORT,//,5HABORT//)
(G10.3,3("READ"E12.4)) = (G10.3,"READ"E12.4,"READ"E12.4,"READ"E12.4)
```

SPECIFICATION INTERRELATIONSHIPS

Two or more specifications (E9.3,I6) in a FORMAT statement are concatenated: Data 12.3 and –30303 produces

Δ.123E+02	-30303

The *n*X edit specification (E9.3,4X,I6) can insert blank spaces between fields: The same data produces

Δ.123E+02	ΔΔΔΔ	-30303

Or the / edit specification (E9.3/I6) places each field on a different line: The same data produces

Δ.123E+02
-30303

Nesting

The group of format and edit specifications in a FORMAT statement can include one or more other groups enclosed in parentheses (in this text, called "group(s) at nested level *x*"). Each group at nested level 1 can include one or more other group(s) at nested level 2; those at level 2 can include group(s) at nested level 3; those at level 3 can include group(s) at level 4:

(E9.3,I6,(2X,I4))	One group at nested level 1.
(T12,"PERFORMANCES"3/(E10.3,2(A2,L4)))	One group at nested level 1, one at nested level 2.
(T5,5HCOSTS,2(M10.3,(I6,E10.3,(A2,F8.2))))	One group at nested level 1, one at level 2, one at level 3.

A FORTRAN READ or WRITE statement references each element of a series of list elements; the Formatter scans the corresponding FORMAT statement to find a field descriptor for each element. As long as a list element and field descriptor pair occurs, normal execution continues. Formatter execution continues until all list elements have been transmitted.

Unlimited Groups

If a program does not provide a one-to-one match between list elements and field descriptors, Formatter execution continues only until all list elements have been transmitted. If there are fewer written field descriptors than list elements, format specification groups at nested level 1 and deeper are used as "unlimited groups." After the effective rightmost field descriptor in a FORMAT statement has been referenced (see "Repeat Specifications—For Field Descriptors"), the Formatter performs three steps:

1. The current record is terminated: on output, the current field is completed, then the record is transmitted; on input, the rest of the record is ignored.
2. A new record is started.
3. Format control (field descriptor interpretation) is returned to the repeat specification for the rightmost specification group at nested level 1. Or, if there is no group at level 1, control returns to the first field descriptor (and its repeat specification) in the FORMAT statement.

NOTE: In any case, the current scale factor is not *changed until another scale factor is encountered (see "Scale Factor").*

EXAMPLES:

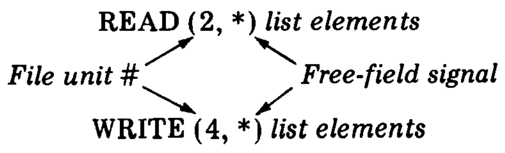
(I5,2(3X,F8.2,8(I2)))	Control returns to 2(3X.F8.2,8(I2))
(I5,2(3X,F8.2,8(12I2)),4X,(I6))	Control returns to (I6)
(I5,3X,4F8.2,3X)	Control returns to (I5,3X,4F8.2,3X)
("HEADER" /3(E10.2))	Control returns to 3(E10.2) to produce:

```
HEADER
|   E10.2    |    E10.2    |    E10.2    |
|<-- E10.2 -->|<-- E10.2 -->|<-- E10.2 -->|
|   E10.2    |    E10.2    |    E10.2    |
```

FREE-FIELD INPUT/OUTPUT

Free-field input/output is formatted conversion according to format and/or edit control characters imbedded in the data. That is, the Formatter converts data from or to external ASCII character form without using FORMAT statements. For free-field inputs, format and/or edit control characters are imbedded in the external data fields. For free-field outputs, predefined field and edit descriptions are used.

For free-field input/output, FORTRAN READ or WRITE statements are written with an asterisk instead of a FORMAT statement identifier:

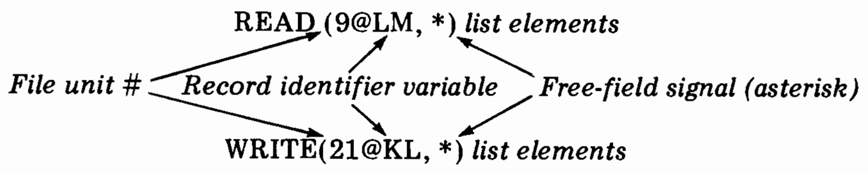


```
READ (2, *) list elements
WRITE (4, *) list elements
```

For free-field input/output to or from disc devices (see "Disc Input/Output," earlier in this section), READ or WRITE statements in a FORTRAN program are written:

For sequential access: As described on the preceding page for free-field input/output.

For direct access:



```
READ (9@LM, *) list elements
WRITE(21@KL, *) list elements
```

Free-Field Control Characters

Special ASCII characters embedded in the external data fields control free-field input:

Character(s)	Function
(Blank space) or , (comma) or any ASCII character not part of the data item.	Data item delimiter (terminator)
/ (slash)	Record terminator (when not part of a character string data item)
+ (plus) or – (minus)	Sign of data item
. (period)	Define the beginning of the fraction subfield of the data item
E or + or – or D	Define the beginning of the exponent subfield of the data item
% (percent)	Define the data item as octal (not decimal)
" . . . "	A character string enclosed by quotation marks; to be input to a FORTRAN/3000 type character variable.
<< . . . >>	A character string enclosed by << and >>; the characters are a comment only for the external record; the string and symbols are ignored on input

Free-Field Input

Six data types can be input to free-field conversion: octal, integer, floating-point (real), double-precision floating point, and character string. Numeric data types can be mixed freely with numeric list elements. For example, an integer data intem can be input to a floating-point list element; the Formatter converts the integer to floating-point form and stores the double-word result.

All rules for input to numeric and alphameric conversions (see "Field Descriptors") apply.

A character string item, however, must be input only to a character string list element; if not, SOFTERROR' message FMT: STRING MISMATCH: occurs and Formatter execution is aborted.

DATA ITEM DELIMITERS

A **data item** is any numeric or character string field occurring between data item delimiters. A **data item delimiter** is a comma, *a blank space*, or any ASCII character that is not a part of the data item. The initial data item need not be preceded by a delimiter; the function of a delimiter is to signal the end of one data item and the beginning of another.

Two commas with no data item in between indicate that no data item is supplied for the corresponding list element, and the previous contents of that list element are to remain unchanged. Any other delimiter appearing two or more consecutive times is equivalent to one delimiter.

NOTE: Do not include a "no-data" field in a series of free-field data inputs. For example, a remark field such as REMARK: I=1234 IS CORRECT *will not prevent the digits 1234 from being interpreted as a free-field data item.*

DECIMAL DATA

Decimal data items are written in any of the forms described under "Field Descriptors," except the monetary or the numeration forms. Imbedded commas or the dollar sign are data item delimiters.

NOTES:
1. *Leading, imbedded, or trailing blanks or commas, $, etc., are data item delimiters.*
2. *All integer inputs have an implicit decimal point to the right of the last (least significant) digit.*
3. *The exponent field input can be any of several forms:*

+e	*+ee*	*Ee*	*Eee*	*De*	*Dee*
–e	*–ee*	*E+e*	*E+ee*	*D+e*	*D+ee*
		E–e	*E–ee*	*D–e*	*D–ee*

where e is an exponent value digit.

OCTAL DATA

Octal data items are written

$\%i_1 \ldots i_n$

where

$i_1 \cdots i_n$	=	the octal integer digits
n	=	the number of octal digits (maximum: 6)
%		is the octal data identifier

Non-octal digits are delimiters. The largest number allowed is 177777_8. If n is greater than 5, the first (most significant) digit must be 0 or 1.

CHARACTER STRING DATA

A character string data item is any series of ASCII characters, including blank spaces, enclosed in quotation marks. Any one or more of those characters can be a quotation mark if signaled by an adjacent quotation mark.

The corresponding list element must be of type CHARACTER in FORTRAN/3000 (or type BYTE ARRAY in SPL/3000) of a specified string length. If the number of characters in the data item is greater than the length attribute *n* of the list element, *n* characters are transmitted and the remaining characters are ignored. If there are fewer characters than *n*, all characters of the data item are transmitted, left-justified in the list element, followed by trailing blanks.

If an end-of-record condition occurs before the terminating quotation mark of a character string data item, the Formatter assumes the data item is continued in the next record and resumes transmission with the first character of the next record.

RECORD TERMINATOR

The character / (slash), if not part of a character data item, terminates the current record and delimits the current data item. If this occurs before all list elements have been satisfied, the remainder of the current record is skipped and transmission resumes with the first character of the next record.

LIST TERMINATION

If an end-of-record condition occurs without the record terminator /, the effect is to end the list of variables. Any list elements not satisfied are left unchanged.

Free-field Output

Five data types can be output under free-field conversion: integer, floating-point (real), double precision floating-point, and character string. All output is compatible with the requireements of free-field input: it does not require external changes to be input using free-field conversion.

1. Integer data items are output under the I6 field description.
2. Double-integer data items are output under the I11 field description.
3. Floating-point data items are output under the G12.6 field description.
4. Double-precision floating-point data items are output under the G17.11 field description.
5. Character string data items are output under the " . . . " edit description; the adjacent quotation marks rule for included quotation marks is used. (If a quotation mark is included in the string, a double quote is output.)

DATA ITEM DELIMITER

Each field in the output record is delimited by one blank space.

RECORD TERMINATORS

If the width of a current numeric data item is too great for the remainder of a current record, a record terminator character / is output, and a new record is started with the first character of the data item.

If a character string data item overlaps record boundaries, subsequent records are output (without record terminator slashes) until the entire character string has been transmitted.

ACCEPT/DISPLAY

FORTRAN/3000 ACCEPT and DISPLAY are alternate applications of free-field input and output. They are invoked by program statements such as

```
ACCEPT INT,ARRAY,LETR        or        DISPLAY INT,ARRAY,LETR
```

where INT, ARRAY, and LETR are typical list elements. The key words ACCEPT and DISPLAY are equivalent to READ(*i*, *) and WRITE(*o*, *), where *i* is the MPE/3000 file system name of a standard input device $STDIN and *o* is the name of a standard output device $STDLIST, and * is the free-field signal.

Transmissions by ACCEPT and DISPLAY conform to the descriptions given for free-field input and output, with one exception: the Formatter determines if the standard output device to be used is an interactive terminal (such as a teleprinter or a CRT keyboard/display); if the device is an interactive terminal, the ACCEPT routine prints a prompt character, ?, before accepting inputs.

CORE-TO-CORE CONVERSION

Conversions between external ASCII records and a list of variables use an input/output (I/O) buffer allocated to the Formatter. Core-to-core conversions, on the other hand, transfer to and from user-defined buffers (byte arrays). The user can manipulate the data, transmit it to or from external records, or return it to the original location or any other location.

To invoke core-to-core conversion FORTRAN READ and WRITE statements are written:

READ (*v*,*f*) *list elements*   or   WRITE (*v*,*f*) *list elements*

where

v = a character simple variable or a character array element

f = the FORMAT statement identifier

Core-to-core conversions are subject to the same rules, restrictions, and interactions as formatted or free-field conversions to and from external records, with the following exceptions:

1. **Any signal to terminate the current record and start a new record (such as edit specification /, or free-field record terminator /, or the end of an unlimited group sequence) is taken to be an error; SOFTERROR' message FMT: BUFFER OVERFLOW occurs.**
2. If an end-of-record condition occurs before either a terminating quotation mark (") or a close comment symbol (>>) is encountered in free-field data, SOFTERROR' message FMT: BUFFER OVERFLOW occurs.

UNFORMATTED (BINARY) TRANSFER

Data can be transferred to and from disc or tape files in internal representation (binary) form without any conversion. Such transfers are faster and occupy less space than formatted data transfers.

Two types of access to files on disc devices are available through the MPE/3000 file system: sequential or direct. Either type can be established through the MPE/3000 file intrinsic FOPEN.

When binary /sequential access is used, the READ or WRITE statements of a FORTRAN program are written without a FORMAT statement identifier.

EXAMPLES:

$$\begin{tikzcd}
\text{READ (8) \textit{list elements}} \\
\textit{File unit \#} \arrow[u] \arrow[d] \\
\text{WRITE (12) \textit{list elements}}
\end{tikzcd}$$

When binary /direct access is used, the READ or WRITE statements of a FORTRAN program are written with an integer simple variable for the record identifier and without a FORMAT statement identifier.

EXAMPLES:

$$\begin{tikzcd}
& \text{READ (8@IV) \textit{list elements}} & \\
\textit{File unit \#} \arrow[ur] \arrow[dr] & & \textit{Record identifier variable} \arrow[ul] \arrow[dl] \\
& \text{WRITE (12@KR) \textit{list elements}} &
\end{tikzcd}$$

When the file is opened (through the MPE/3000 file intrinsic FOPEN), the record size can be left at the system default value 128, or the user can specify a different size.

In sequential access, as many records as needed are used in sequence until the entire list of elements has been transferred.

> *NOTE: If the storage required exceeds the size of the record, transfer continues into the next record; this usually leaves part of that next record unused.*

In direct access, record access is terminated by the last element in the list. Any unused portion of the record just terminated is ignored.

If the storage required by all the elements in the list exceeds the record size, SOFTERROR' message FMT: DIRECT ACCESS OVERFLOW occurs.

Matching List Elements

The binary transfer user must match list elements between corresponding READ and WRITE statements of a FORTRAN program. For example, if a list of elements is transferred to a disc, any corresponding return of the data to internal storage must do so to a list that matches each element by type and dimensions and by order of appearance in the list. The simplest method is to use the same element labels for input and output, if possible.

> *NOTE: Under binary /direct access, the Formatter begins each new list element output at a word boundary. If the list element is, for example, a byte array of an odd number of bytes, one byte of the record will not be used.*

SECTION X

FORTRAN File Facility

Every peripheral transmission or storage device is linked to a file through the file facility of the operating system. Each file takes on the attributes of the hardware device associated with it. FORTRAN input or output statements reference specific hardware devices (such as teleprinters or card readers) by referencing the associated file. The FORTRAN compiler translates the FORTRAN input/output statements into requests for manipulations of the files referenced in the input/output statement. Manipulation of the files (and hence the peripheral device) is handled by non-FORTRAN language routines called intrinsics, which are part of the operating system. A FORTRAN user need only reference the appropriate file for a data transmission. The operating system handles the transfer for the user.

Files are referenced in FORTRAN input/output statements by using integer constants or integer simple variables with values in the range of 1 to 99, inclusive. Only those values are used that correspond to an existing file in the system. The same number can be used in more than one input/output statement; the same file is referenced in each case. File numbers need not be declared available before they are used. The FORTRAN compiler assumes the existence of any file specifically referenced by an integer constant in the control part of an input/output statement.

If the file reference in the control part of an input/output statement appears as an integer simple variable, the file number represented by the variable must appear in a FILE compiler control record at the beginning of the program unit (see Section XI) if it does not also appear explicitly in some other I/O statement. The file facility opens a file when it is first referenced, and closes it when the program terminates.

The DISPLAY statement implicitly declares and references file 6. The ACCEPT statement references file 6 for prompting and file 5 for the user's response (see Section VII).

STANDARD INPUT AND LIST FILES

The integer values used as file names in the source program are converted to names acceptable to the operating system file facility. The name for any file is created by concatenating the characters FTN with the two-character alphameric representation of the integer name. For example, file 8 is FTN08, file 10 is FTN10, etc.

By default unit 5 is the standard input file, typically a card reader for the batch user, or a teleprinter for the interactive user. Unit 6 is the standard list file, typically a line printer for the batch user or a teleprinter for the interactive user. All other files are scratch files and are considered to be disc files unless otherwise specified by the appropriate operating system command.

CHANGING STANDARD ATTRIBUTES OF FILES

The standard attributes of files used by the FORTRAN programmer may be modified through the use of MPE/3000 :FILE commands. These commands are submitted with the user's source program. Each command contains the file name (FTN06, FTN10, for example) and the list of file attribures which the user desires to change (those attributes not mentioned in the record remain unchanged).

:FILE can store files under the same or any other name (FTN12 can be given the name PRINT and stored elsewhere). At the end of program execution, a file can be saved under another name and the old file can be purged. Any file can be modified through :FILE commands.

DIRECT INTRINSIC CALLS

Since a FORTRAN user can write and execute non-FORTRAN language programs, it is possible to access files directly. This is accomplished by writint direct calls to operating systems intrinsics which manipulate the indicated files. The calls may require actual arguments passed by value (see Section VI). Direct intrinsic calls may be used intirely by themselves or in conjunction with label equation records.

Care must be taken when using direct intrinsic calls for file manipulation. Status information is maintained for all files referenced in FORTRAN input/output statements. Reading from or writing into files by directly accessing intrinsics is safe, but opening or closing a file that is mentioned in a FORTRAN input/output statement within a user's program may invalidate the program's subsequent attempts to reference the file. However at program termination, the system file facility closes all the files mentioned in a user's program, regardless of how they were opened.

Note: :FILE commands and file intrinsics are described in the *HP 3000 Multiprogramming Executive Operating System (03000-90005).*

SECTION XI

Compiler Subsystem Commands

The FORTRAN/3000 compiler reads source code, generates object code, and produces listings through the MPE/3000 file system. FORTRAN/3000 compiles a source program located in the *textfile*, generates object code to the *uslfile*, outputs listings to the *listfile* and performs editing function from a *masterfile* and a *newfile*. The actual files names used are equated through MPE/3000 commands (see *HP 3000 Multiprogramming Executive Operating System (03000-90005).* Table 11-1 shows the formal file designator and the default file designators for FORTRAN/3000 compiler subsystem commands.

Table 11-1. Compiler Subsystem File Names

File	Use	Formal File Designator	Default File Designator
Textfile (source)	Source program, Corrections, Compiler commands	FTNTEXT	$STDIN
Listfile	Output listing	FTNLIST	$STDLIST
Uslfile	Code output	FTNUSL	$NEWPASS/ $OLDPASS[1]
Masterfile	Old copy for edit	FTNMAST	$NULL
Newfile	New copy for edit	FTNNEW	$NULL

The compiler subsystem commands are entered through the *textfile* and take effect only after the compiler has been accessed. In the following description of these commands, brackets are used to enclose optional items, braces are used to enclose required items (one of which must be chosen), and ellipses (...) indicate repeated items.

The basic syntax of subsystem commands is

$*command* [*parameter list*]

The $ must be the first character in the FORTRAN line and immediately be followed by the command name, which must be completely spelled out. The parameter list, separated from the name by at least one blank, may be optional; some commands have parameters, others do not. Parameters are separated from each other by commas. Blanks may be freely inserted between items in the list.

[1]If $OLDPASS exists and is a USL file.

A command can be continued for as many as 19 additional records if the last nonblank character is an ampersand (&). If the last character is an ampersand, the following record must begin with a $. The effect is to concatenate the characters preceding the & with those following the $ of the next record. Command names and parameters must not be broken by an &. The sequence field of each record (source or command) is ignored by the compiler and can be used for sequence numbers (see "EDIT Commands" in this section.

CONTROL COMMAND

The CONTROL command specifies list and compilation options during source program compilation. A CONTROL command can appear anywhere a statement or comment can appear, although some parameters are honored only at special points during compilation. The form for CONTROL is

$CONTROL *parameter list*

The possible parameters are given in Table 11-2.

Table 11-2. CONTROL List and Compilation Options

Parameter	Description
BOUNDS	Requests the compiler to emit code for the dynamic validation of array indices. the compiler checks adjustable as well as fixed-size arrays; it also checks dummy arrays by using information in the DIMENSION statements as opposed to any actual attributes of the arrays associated during execution of the program unit. The BOUNDS parameter is honored only at the beginning of a program unit and is cleared by default unless specified.
CODE	Sends relocatable machine code records from the compilation to the *listfile* after appropriate sections of the source records are sent. The CODE parameter is honored only at the beginning of the program unit and is cleared by default at the beginning of each program unit.
ERRORS=*ddd*	Sets a maximum number of severe errors allowable before the compiler terminates compilation of the program unit (0 ≤*ddd* ≤999). The ERRORS=*ddd* parameter sets the maximum number of severe errors at 50 by default for each program unit compiled.
FILE = positive integer	Declares that any executable FORTRAN program containing the program unit with the FILE record has access to a file with a positive integer as its FORTRAN/3000 file number. The FILE = positive integer option is used for files whose FORTRAN numbers do not explicitly appear in a FORTRAN I/O statement but will be used as the value of an integer simple variable in an I/O statement (see Section X, "FORTRAN/3000 File Facility").

Table 11-2. CONTROL List and Compilation Options (Continued)

Parameter	Description
FIXED	Specifies that the source records following the FIXED parameter appear in fixed-field format (see Section I). The FIXED parameter, honored anywhere in the program unit, is on by default and remains set through the entire compliation unless explicity changed by the FREE OPTION.
FREE	Specifies that the source records following the FREE parameter appear in free-field format (see Section I). The FREE parameter is honored anywhere in the program unit and clears the FIXED option.
INIT	Requests the compiler to emit code to initialize all local simple variables and arrays upon each invocation of the program unit. Arithmetic values are initialized to zero, logical values to FALSE, and character values to all null characters. The INIT option is honored only at the beginning of the program unit and is cleared by default if not specified.
LABEL	Requests a listing of the program units label map following the machine code listing. Addresses are supplied for statement labels and FORMATS stored in the code. The LABEL parameter is cleared by default at the beginning of compilation.
LIST	Sends each source record after editing to *listfile* (see "EDIT Commands" in this section). The LIST parameter is set by default for compilations in batch mode and can appear anywhere in a program unit. Once set, the parameter remains set for the entire compilation unless the NOLIST parameter is used later in the compilation.
MAP	Sends a symbol table dump to the *listfile* following the program unit source records. The MAP parameter is cleared by default at the beginning of compilation.
NOCODE	Clears the CODE parameter if it is on. The NOCODE parameter is set by default at the beginning of compilation.
NO LABEL	Clears the LABEL parameter if it is on.
NOLIST	Sends only offending source records and error message records to the *listfile.* The NOLIST parameter is initially set by default for compilations in interactive mode and can appear anywhere in a program unit. Once set, the parameter remains set for the entire compilation unless the LIST parameter is used later in the compilation.
NOMAP	Clears the MAP parameter if it is on.
NOWARN	Clears the WARN parameter if it is set.

Table 11-2. CONTROL List and Compilation Options (Continued)

Parameter	Description
SEGMENT = segment name	Assigns a specific segment name to the program unit. The SEGMENT = segment name parameter is used by the MPE/3000 loader to determine which code modules are combined into executable program segments. All program units having the same segment name — and only those program units — are placed into the same segment. All program units lacking a segment name are placed into the segment containing the default name SEG'. This option has no effect on BLOCK DATA subprograms since they produce no code.
WARN	Sends offending source records and warning meassages to the *listfile*. The WARN parameter is initially set by default at the beginning of compilation.

PAGE COMMAND

The form for the PAGE command is

```
$PAGE [character string list]
```

The PAGE command ejects the current page of *listfile* to the top of the next page and prints a page heading followed by two blank lines. If the optional character list is present, it appears as the title of this and following page headings until changed by a subsequent PAGE or TITLE command. *Character string list* consists of one or more character strings separated by commas (each string is bracketed with quote marks.) When the list is printed, the quote marks, separating commas and any blanks between strings are deleted and the character strings themselves are concatenated and placed in the heading. One character string in the list cannot be continued from one line to the next, but individual strings in the list can be continued on continuation lines. No page eject occurs if NOLIST is on since no listing is being printed. No page eject takes place if the PAGE command is within the range of an unsatisfied IF command. A PAGE command example is

```
$PAGE "MIDDLE OF", "THE PROGRAM"&

$, "UNIT COMPILATION"
```

TITLE COMMAND

The form for the TITLE command is'

$TITLE [character string list]

The TITLE command is used to print a page heading whenever the top of the page is encountered in *listfile*. The *character string list* is used for subsequent page headings until another PAGE or TITLE command is encountered. If no *character string list* is present, the title field is set to all blanks. *Character string lists* is defined in the same way that the character string list is defined for the PAGE command. TITLE commands can be continued on continuation lines in the same manner as PAGE commands. An example of a TITLE command is

```
$TITLE "THE DATE",&

$"TODAY IS",&

$"10/10/72"
```

SET COMMAND

The form of a SET command is

$$\text{\$SET } [X_n = \begin{Bmatrix} \text{ON} \\ \text{OFF} \end{Bmatrix} [X_n = \begin{Bmatrix} \text{ON} \\ \text{OFF} \end{Bmatrix}] \ldots]$$

where *n* varies from 0 to 9, inclusive.

The SET command sets or clears toggle X_n if no parameters are specified, all ten toggles are cleared. If more than one parameter is given, each parameter must be separated from the next by a comma. All toggles begin the compilation in the cleared (OFF) state. The SET command is used with the IF command to bypass portions of the source file during compilation (see "IF Command," in this section).

IF COMMAND

The form of an IF command is

$$\text{\$IF } [x_n = \begin{Bmatrix} \text{ON} \\ \text{OFF} \end{Bmatrix}]$$

where *n* varies from 0 to 9, inclusive.

The IF command specifies a condition (X_n = ON or X_n = OFF). If the specified condition is false, i.e., if the toggle is not in the state specified, all succeeding source records are ignored by the compiler until another IF command is specified. The only command not ignored during the bypass is the EDIT command. If the specified condition is true, succeeding source records are compiled normally. An IF command with no parameters specified merely terminates the preceding IF command. Any source records ignored by the compiler are listed and sent to *newfile*. Toggles are set using the SET command (see "SET Command" in this section).

EDIT COMMAND

The form for the EDIT command is

$EDIT *parameter* [,*parameter*] ...

where any of the following *parameters* may be specified:

VOID = *sequence number*

SEQNUM = *sequence number*

NOSEQ

INC = *number*

Use of the EDIT command depends on the parameters specified (see "EDIT Parameters" for *parameter* meanings) and the files specified in the MPE/3000 subsystem command :FORTRAN. For a description of the :FORTRAN command and parameters, consult *HP 3000 Multiprogramming Executive System (03000-90005).*

The editing capabilities available are

- Merging correction records with an old master program to produce a new program for compilation
- Checking source record sequence numbers for ascending order
- Bypassing sections of source program
- Renumbering source record sequence numbers

If the :FORTRAN command specifies both *masterfile* and *textfile*, source records from both files are merged, and the record sequence numbers are checked for ascending order. Each sequence number in columns 73 to 80 of the record must either be all blank or greater than the previous sequence number. In merging *masterfile* with *textfile*, one record is read from each file and their sequence numbers are compared. The record with the lower sequence number is compiled and passed to the *newfile*. If the sequence numbers are identical, the record from *textfile* is compiled and passed to *newfile* (if it exists).

By default records set to *newfile* are sent with unchanged sequence numbers. To renumber sequence numbers, use the SEQNUM parameter of the EDIT command. Sequence numbers are checked by the compiler for proper order only if *masterfile* is specified in the :FORTRAN command and *textfile* is not specified. *masterfile* is not specified, sequence numbers are not checked by the compiler.

EDIT command records can contain sequence numbers to indicate placement in the *textfile* but EDIT continuation records must have blank sequence fields. Unlike other source records or commands, EDIT commands are not sent to *newfile.*

Edit Parameters

The EDIT command can contain VOID, SEQNUM, NOSEQ, or INC parameters. The parameters and their meanings appear below:

The VOID parameter form is

VOID = *sequence number*

When the VOID parameter appears in an EDIT command, the compiler bypasses all *masterfile* records with a *sequence number* less than or equal to the *sequence number* in the VOID parameter. The *sequence number* can be specified either as a number, The compiler left-fills the number with zero (0) digits, or strips the "string" of the quote marks and left-fills the field with blank characters to achieve eight characters. The first form is compatible with sequence numbers as generated by the SEQNUM parameter (see below); the second form is compatible with sequence numbers generated by the HP 3000 Text Editor program.

The form for the SEQNUM parameter is

SEQNUM = *sequence number*

The SEQNUM parameter renumbers the following source records sent to *newfile* starting with the *sequence number* specified in SEQNUM. If the INC parameter (see below) is specified, each record *sequence number* is incremented by the value associated with INC. If INC is not specified, *sequence number* is incremented by the default value 1000 for each succeeding record.

The form for the NOSEQ parameter is

NOSEQ

NOSEQ indicates that following source records retain their current sequence numbers. If SEQNUM = *sequence number* is not specified, the NOSEQ condition occurs regardless of whether NOSEQ is specified.

The form for the INC parameter is

INC = *number*

number indicates the value by which each source record *sequence number* is incremented when the SEQNUM parameter is specified. INC is ignored if *newfile* is not specified or if the last SEQNUM parameter was overridden by a NOSEQ.

The form for the FIXED parameter is

FIXED

The FIXED parameter following an EDIT command informs the compiler that the source records in the *textfile* are in fixed-field format (sequence field is located in columns 73-80 of the source record). If no *masterfile* exists, FIXED is ignored.

For form for the FREE parameter is

FREE

The FREE parameter following an EDIT command informs the compiler that the source records in the *textfile* are in free-field format (sequence field is located in columns 73-80 of the source record). If no *masterfile* exists, FREE is ignored.

When a record is read from the *textfile*, the compiler must locate the sequence field to determine when the record is to be merged with the *masterfile*. At the beginning of compilation of after an EDIT command specifying FIXED, the compiler takes characters 73-80 of the record as the sequence field. Following an EDIT command specifying FREE the compiler takes the sequence field to be the first character of the record up to (but not including) the first blank. The compiler uses only the last eight characters and prefixes the sequence string with ASCII zero characters if less than eight. A blank in column one indicates an all blank sequence field.

When the record from the *textfile* is merged with the *masterfile* if the mode (FIXED or FREE) is the same as the mode of source records in the *masterfile*, then the textfile record is used as is. If the mode differs, the *textfile* record is converted to the *masterfile* mode. A line read as FIXED is converted to FREE by moving the sequence field (columns 73-80) to the beginning of the line, inserting a blank following the sequence field, and following that with characters originally in columns 1 through 71 of the line. Column 72 of the line is lost since free-field records contain a maximum of 71 characters following the sequence field. A line read as FREE is converted to FIXED by removing characters from 1 to 72 and using the last eight characters of the sequence field (remember the blank is not part of the field) as characters 73 - 80.

Note that the records in *textfile* which will be converted should be written in the mode of the *masterfile* except for the sequence field. For instance, comment lines merged into a fixed field program must use the letter C (in the appropriate place in the line) instead of "#", even though the line is entered into the *textfile* in free-field mode.

TRACE COMMAND

The form for the TRACE command is

!TRACE [*program unit*] ;*identifier, identifier, ...*

The TRACE command specifies variables, arrays, labels, and other program elements (*identifiers*) to be monitored by the HP 3000 Symbol Trace program (TRACE/3000) during program execution. *program unit* is the name of the program unit to which the *identifiers* belong, (if omitted, the compiler uses the name MAIN'). The compiler inserts calls to the TRACE/3000 subprogram at appropriate points in the object code generated during compilation.

$TRACE records must appear before the first FORTRAN statement of the program unit to which the $TRACE records apply. For a multiprogram unit compilation, $TRACE records can appear before any program unit preceding the affected one, thus allowing all $TRACE records to be grouped before the first program unit in a multiunit compilation. For further information on the TRACE/3000 program, consult *HP 3000 Symbol Trace (03000-90015).*

APPENDIX A

Non-FORTRAN Program Units

Any non-FORTRAN language program unit may be used as part of an executable FORTRAN/3000 program, provided the program unit has a calling sequence and effects of execution compatible with FORTRAN. Conversely, a FORTRAN/3000 language subprogram can be used by a program written in some other language, as long as its use is compatible with the calling program's requirements.

All arguments of a subprogram written in FORTRAN/3000 must be passed by reference. A function reference or CALL statement prepares a list of addresses for the actual arguments associated with the call. In a function, space is allocated immediately before the address list for the value associated (returned) with the function name after execution. A subprogram written in FORTRAN/3000 deletes the actual argument addresses from its data space after it is executed. Any FORTRAN/3000 program referencing a non-FORTRAN language subprogram expects that subprogram to delete its actual parameter addresses. FORTRAN/3000 also expects FORTRAN/3000 and non-FORTRAN function subprograms to retain the value returned in the subprogram data storage space.

Non-FORTRAN program units may require actual arguments passed by value. No FORTRAN/3000 program unit allows arguments passed by value, but a calling program can pass arguments by value to non-FORTRAN subprograms if they require it (see Section II), "Dummy and Actual Argument Characteristics."

SPL/3000 PROGRAMS

SPL/3000 (System programming Language for HP/3000) can be used for non-FORTRAN subprograms; SPL/3000 uses the same calling sequence as FORTRAN/30000. SPL/3000 function and subroutine subprograms are invoked in the same manner in an SPL/3000 source program. A function written in FORTRAN/3000 is called in an SPL/3000 program in the same manner.

SPL/3000 programs do not accept complex values as do FORTRAN programs, while SPL/3000 includes the use of double integers and FORTRAN/3000 does not. Double precision real numbers in FORTRAN/3000 are called long real in SPL/3000 although their use is compatible. FORTRAN statement labels are not useful to an SPL/3000 program, and SPL/3000 statement labels are not allowed in FORTRAN; they cannot be passed as actual arguments between one language and the other. SPL/3000 also includes the use of pointer variables, which is useful only if the data space in FORTRAN/3000 programs corresponds to the SPL/3000 data space. Pointers are passed by value from FORTRAN/3000 to SPL/3000, and by reference from SPL/3000 to FORTRAN/3000.

Arrays are passed between SPL/3000 and FORTRAN/3000 programs by supplying an array element as the actual parameter. The first element of an SPL/3000 array is element zero, while the first element of a FORTRAN/3000 array is element 1. SPL/3000 allows only one-dimensional arrays, so any multidimensional arrays passed from FORTRAN/3000 to SPL/3000 are linearized according to the array successor function (see Section III). A character value in FORTRAN/3000 corresponds to a byte value in SPL/3000.

SYSTEM INTRINSICS

System intrinsics can be invoked from FORTRAN/3000 programs using SPL/3000 language subprograms, subject to the restrictions just mentioned. The system intrinsics and their uses (including calling sequences) are described in *HP 3000 Multiprogramming Executive Operating System (03000-90005).*

APPENDIX B

FORTRAN/3000 and ANSI Standard FORTRAN

FORTRAN/3000 conforms to the American National Standard Institute's Standard for FORTRAN (X.39 — 1966). To provide a more powerful programming tool, FORTRAN/3000 extends beyond the Standard and in some minor cases, places restrictions on the Standard to conform with the HP 3000 computer system architecture. A brief description of each extension or restriction appears below. Numbers in the "Standard Reference" column are references to the appropriate text in the Standard (X3.9 — 1966).

Standard Reference	Comments
3.	Program preparation from terminals can occur in a free-field as well as a fixed-field format.
3.1	FORTRAN/3000 uses a 128-character USACII 8-bit standard character set. All printing characters can appear in Hollerith and string values. Some of the control characters are reserved for special purposes (such as carriage return or line feed).
3.2	End lines can appear with a statement label preceding.
3.5	Symbolic names consist of as many as 15 characters instead of just 6.
4.	Character-type data can be used in FORTRAN/3000 programs to facilitate data manipulation.
4.2	Logical data can be manipulated as 16-bit binary masks in addition to their function as true/false data.
5.1.1	Constants of all types can be specified in more than one way by using octal values, partial-word designators, etc. Character constants in the form of string or Hollerith values can also be used.
5.1.3	FORTRAN/3000 allows arrays of up to 255 dimensions instead of just three dimensions allowed by the Standard. Subscript expressions are any linear expressions.

Standard Reference	Comments
5.3	The IMPLICIT statement can be used to generalize the data type associated with the first letter of an identifier to include integer, real, double precision, complex, or character. Function subprograms can determine their type through a Type statement within the subprogram defining unit.
6.	Expressions of type character can be used to facilitate the use of character data.
6.1	Expressions can be created using primaries of different types. In assignment statements, the resulting expression value type is converted to the type of the identifier on the left side of the assignment indicator.
	In exponentiation, constructs such as A**B**B are allowed (without the need for parentheses) and can use powers and bases of differing types. No base can be raised to a complex power, however.
	Partial-word designators allow manipulation of the subparts of integer values.
6.3	FORTRAN/3000 includes an "exclusive OR" operator. The other relational operators are generalized. Expressions of type integer, real, or double precision can appear on one side of a relational operator with an expression of type integer, real, or double precison on the other side. Complex expressions can appear between equal (.EQ.) or not equal (.NE.) signs only.
7.1.1	The identifier to the left of the assignment operator in an assignment statement need not be of the same type as the expression on the right of the operator. The expression value type is converted to the identifier type prior to assignment. Partial-word designators can be used to assign parts of integer or logical variables. Character-type assignment statements can be used providing the left and right-hand parts are of type character.
	Label data and integer data are mutually exclusive. A variable of the same name can be assigned values of both types without ambiguity.
7.1.2.1	The assigned GO TO statement does not require a list of labels. The computed GO TO can use a linear expression for its index for selecting the transfer statement.
7.1.2.3	The dependent statement of a logical IF statement can be ancther logical IF statement.
7.1.2.4	A label can be used as an actual argument in a CALL statement to allow alternative return points following execution of the subroutine referenced by CALL.

Standard Reference	Comments
7.1.2.5	An optional exit label can be included in the RETURN statement to return to one of the calling program unit's statements whose label appears as an actual argument to the subroutine containing the RETURN.
7.1.2.7	STOP or PAUSE statements use a decimal integer for identification rather than an octal integer.
7.1.2.8	FORTRAN/3000 supports the concept of extended DO ranges, as discussed in this manual.
7.1.3	Direct-access files can be referenced in FORTRAN/3000 input/output statements. These statements allow extended format and error recovery capabilities'
7.2.1.1	Adjustable array declarators can be used for local arrays in subprograms to select different size arrays for each activation of a subprogram.
7.2.1.3	Since FORTRAN/3000 executes on a 16-bit word machine, integer and logical values require one word of computer memory, real values two, double precision three, complex four. Character data uses half-word storage units.
7.2.1.6	Type statements for type character is available.
7.2.2	The DATA statement in FORTRAN/3000 extends beyond the Standard as described in this manual.
7.2.3	Additional editing types other than those described in the Standard are available in FORMAT statements.
8.1	The defining statement of a statement function can be any expression of the appropriate type. The expression can include array elements.
8.2	FORTRAN/3000 includes a larger set of intrinsic functions than listed in the Standard.
8.3	Recursion in function subprogram defintion is allowed. The type of actual arguments in a function reference has been expanded. Argument are all passed by reference rather than value.
8.4	Subroutines can be defined recursively and can be called with the same actual arguments types as function subprograms.
8.5	Block data subprograms can be given a name.

Standard Reference	Comments
9.	Main program units can be given a name.
10.2	A variable that is defined is always available on the first and second level. For instance, an integer simple variable that is used for both label values and integer values. FORTRAN/3000 never confuses the two values. Any variable appearing in a DATA or COMMON statement remains defined until it is explicitly redefined.

APPENDIX C

FORTRAN/3000 and Previous Versions of HP FORTRAN

FORTRAN/3000 attempts to correspond to previous versions of Hewlett-Packard FORTRAN whenever possible. However, differences between the 2100 family and the 3000 family of computer requires that some differences exist. The following differences are deletions of certain aspects of HP 2100 FORTRAN.

- Octal constants are no longer represented by a B suffix following the constant but are prefixed with %.
- Array variables must explicitly reference all subscripts. Previously, 2100 FORTRAN filled in any omitted subscripts with 1's.
- An array declarator for the same array may not appear in both a DIMENSION and COMMON statement within the same program unit.
- An arithmetic IF statement must always include three statement labels, not just two.
- The logical IF statement cannot be followed by a pair of statement labels in place of an executable statement.
- An END line can contain no other nonblank characters other than a statement label followed by the characters E, N, and D.
- Index expressions such as subscripts and computed GO TO indices cannot evaluate to a complex value.
- Hollerith constants cannot be used in place of integer constants or expressions.
- Statement function names and intrinsic function names cannot be passed as actual arguments for a dummy function name.
- Comments cannot separate a continuation line from its predecessor.

The following differences constitute modifications of various 2100 FORTRAN aspects.

- Intrinsics cannot be passed as actual arguments while Basic External Functions can.
- The @ and K format editing phrases are supplanted by the O editing phrase.
- Character strings appearing as free-field data are enclosed in double quotes (") or

APPENDIX D

Error and Warning Messages

During compliation of FORTRAN/3000 source programs, the compiler prints error messages and warning messages to indicate conditions such as illegal syntax or to warn of marginal conditions which may cause improper execution of the compiler-generated object code. When such a condition occurs, the compiler prints a message which includes a number and a brief explanitory text.

Table D-1 lists all the messages that may occur; Table D-2 includes explanation for error conditions, and Table D-3 outlines warning conditions.

ERROR MESSAGES

If an error condition occurs, the compiler outputs the message

*** ERROR *nnn* *** *message text*

where *nnn* is the message number and *message text* is a brief description of the error condition. Error conditions are nonrecoverable. The compiler flushes any code generated for the current program unit and attempts to compile the next program unit. Table D-2 below describes compiler action taken for specific error conditions. The number indicates the error condition which prompted the message while the "Compiler Action" column describes what action the compiler takes.

WARNING MESSAGES

If a warning condition occurs, the compiler outputs the message

** WARNING *nnn* ** *message text*

where *nnn* is the message number and *message text* is a brief description of the warning condition. Warnings do not inhibit successful compilation of a program unit or the emission of object code by the compiler. Conditions indicated by the message can cause program execution errors if not corrected and if the source code is not re-compiled. Table D-3 describes compiler action taken for specific warning conditions. The number indicates the warning condition which prompted the message while the "Compiler Action" column describes what action the compiler takes.

ERROR POSITION INDICATION

Error and warning messages relate to the source code in one of four ways, depending on the type of message and time the compiler prints the message.

During syntax scanning, the compiler usually prints an up arrow (↑) at or to the right of the error position in the text. Sometimes (as in the case of an illegal character) the compiler prints the arrow under the offending line, while other times the compiler waits until the entire statement is printed before indicating an error. In this case, the compiler prints error indicators refering to previous lines of the statement, following the last line of the statement. If the compiler is not producing a source listing, the compiler prints the last line examined before printing the error message.

Some error messages do not refer to any particular part of the program (e.g., TOO MANY TRACE SYMBOLS) and have no error indication. The user solution is not tied to one symbol or statement.

In some cases such as solving equivalences, the compiler prints an error message after reading the entire program unit. The compiler indicates which statement is being processed by using the form

AT *statement number* + *offset*

statement number is the label of the offending statement or the closest preceding statement label to the offending statement if the offending statement is not labeled. *offset* is a count of the number of statements the offending statement is from the label preceding it. These messages also include the name of the variable being processed when the compiler noticed the error.

Error messages such as TRACE SYMBOL NOT FOUND include a variable name or a label name to indicate the area of the program in error since the error depends upon omitted statements or statement parts.

Table D-1. FORTRAN/3000 Warning and Error Messages

0	COMPILATION TERMINATED --
1	NON-DIGIT IN LABEL FIELD
2	CONTINUATION LINE HAS NON-BLANK LABEL FIELD
3	SYMBOLIC NAME EXCEEDS 15 CHARACTERS
4	EXPECTED A COMMA
5	EXTRANEOUS COMMA
6	EXPECTED A ':'
7	UNEXPECTED CHARACTER
8	UNEXPECTED '-'
9	UNEXPECTED COMMA
10	UNEXPECTED ')'
11	EXPECTED AN INTEGER
12	EXPECTED A '.'
13	EXPECTED A 'P'
14	UNEXPECTED 'P'
15	NESTING EXCEEDS 5 LEVELS
16	EXTRANEOUS SOURCE
17	MULTIPLE SEGMENT NAMES
18	SYMBOLIC NAME REDUNDANTLY TYPED
19	SYMBOLIC NAME REDUNDANTLY EXTERNALLED
20	EXTRANEOUS EQUATE GROUP
22	EXTRA INITIAL VALUES
23	EXTRANEOUS DATA ITEM
24	NO INITIAL VALUES
25	INITIAL VALUE TRUNCATED
26	STATEMENT FUNCTION DUMMY USED AS NON-SIMPLE VARIABLE
27	EXPRESSION VS. NON-EXPRESSION ARGUMENT
28	SUBPROGRAM VS. NON-SUBPROGRAM ARGUMENT
29	SUBROUTINE VS. FUNCTION ARGUMENT
30	ARGUMENT TYPE INCONSISTENT
31	SUBPROGRAM NAME NOT EXTERNALLED
32	ARGUMENTS OF NESTED REFERENCE NOT CHECKED
33	INTRINSIC NAME CONVERTED TO SIMPLE VARIABLE
34	STATEMENT CANNOT BE REACHED
36	RETURN OR STOP INSERTED
38	TOO MANY CONTINUATION LINES
39	EXPECTED CONTINUATION LINE
40	EXPECTED COMPILER CONTROL KEYWORD
41	EXPECTED SYMBOLIC NAME
42	NOT ACCEPTABLE AT THIS POINT
43	IMPROPER STATEMENT LABEL
44	TRACE SYMBOL NOT FOUND --
45	INTRINSIC IN TRACE RECORD
46	EXPECTED A '='
47	SEQUENCE FIELD TOO LONG
48	OUT OF SEQUENCE --
49	TITLE TOO LONG
50	PROGRAM UNIT ABORTED --
51	EXPECTED A '('
52	EXPECTED A ')'
53	EXTRANEOUS ')'

Table D-1. FORTRAN/3000 Warning and Error Messages

55	EXPECTED A ']'
56	EXTRANEOUS ']'
58	EXPECTED ASSIGNMENT OPERATOR
59	EXPECTED A '/'
60	EXPECTED A QUOTE
61	EXPECTED A '\'
63	IMPOSSIBLE CONTEXT FOR '.'
64	IMPOSSIBLE CONTEXT FOR '%'
65	IMPOSSIBLE CONTEXT FOR '$'
67	SYMBOLIC NAME EXCEEDS 255 CHARACTERS
68	NUMBER EXCEEDS 255 CHARACTERS
69	STRING LITERAL EXCEEDS 255 CHARACTERS
70	HOLLERITH LITERAL EXCEEDS 255 CHARACTERS
71	HOLLERITH LITERAL TOO SHORT
72	NON-OCTAL DIGIT
73	EXPECTED FIELD WIDTH
74	EXPECTED FIELD VALUE
75	PACKED NUMBER OVERFLOW
76	INTEGER CANNOT BE 0
77	INTEGER EXCEEDS CONTEXTUAL LIMITS
78	INTEGER OVERFLOW
79	EXPECTED EXPONENT VALUE
80	FLOATING LITERAL UNDER/OVERFLOW
81	IMPROPER COMPLEX LITERAL
84	MISSING END LINE
85	UNEXPECTED CONTINUATION LINE
87	RESERVED TOKEN NOT RECOGNIZED
88	CANNOT RECOGNIZE KEYWORD
89	CANNOT CLASSIFY STATEMENT
91	STATEMENT OUT OF POSITION
92	DUMMY NAME NOT UNIQUE
93	IMPROPER DUMMY ARGUMENT
94	ARGUMENT ADDRESSIBILITY EXCEEDED
95	TOO MANY ALTERNATE RETURNS
96	IMPROPER TYPE CONSTRUCT
97	IMPROPER INITIAL LETTER CONSTRUCT
99	SUBROUTINE CANNOT BE TYPED
100	SYMBOLIC NAME TYPED INCONSISTENTLY
102	DYNAMIC BOUND DIMENSIONED
103	PROCEDURE DIMENSIONED
104	ARRAY REDUNDANTLY DIMENSIONED
105	EXPECTED BOUND
106	ARRAY EXCEEDS 32767 ELEMENTS
107	NUMBER OF BOUNDS EXCEEDS 255
108	DYNAMIC STRUCTURE IN COMMON
109	DUMMY NAME IN COMMON
110	ITEM IN COMMON TWICE
111	COMMON BLOCK NAME ALSO PROCEDURE NAME
112	PROCEDURE NAME IN COMMON
113	CHARACTER FUNCTION HAS DYNAMIC LENGTH
114	SIMPLE VARIABLE OR ARRAY EXTERNALLED
116	DYNAMIC STRUCTURE IN EQUATE
117	DUMMY NAME IN EQUATE

Table D-1. FORTRAN/3000 Warning and Error Messages

118	PROCEDURE NAME IN EQUATE
120	DYNAMIC STRUCTURE IN DATA
121	DUMMY NAME IN DATA
122	PROCEDURE NAME IN DATA
123	COMMON ITEM IN DATA
124	EXPECTED INITIAL VALUE
125	INITIAL VALUE TYPE IMPROPER
126	UNARY SIGN REQUIRES ARITHMETIC LITERAL
127	NUMBER OF SUBSCRIPTS <> NUMBER OF BOUNDS
128	ARRAY EXCEEDS 32767 WORDS
129	SUBSCRIPT VALUE NOT IN ARRAY
131	LOCAL ADDRESSIBILITY EXCEEDED
132	DYNAMIC BOUND NOT DUMMY INTEGER
134	DATA BLOCK TOO LARGE
135	COMMON BLOCK TOO LARGE
136	COMMON FXTENDED FORWARD
137	EQUATE BLOCK TOO LARGE
140	WORD STRUCTURE ALIGNED ON BYTE BOUNDARY
141	DATA BLOCK ITEM EQUATED TO COMMON BLOCK ITEM
142	TWO COMMON BLOCKS EQUATED
143	INCONSISTENT EQUATE
144	SIMPLE VARIABLE HAS SUBSCRIPT
145	EXPECTED STATEMENT LABEL
147	DUPLICATE LABEL
148	UNRESOLVED LABEL REFERENCE --
149	FORMAT REFERENCE TO NON-FORMAT
150	EXECUTABLE REFERENCE TO NON-EXECUTABLE STATEMENT
153	SUBROUTINE USED AS PRIMARY
154	EXPECTED ARITHMETIC PRIMARY
155	NON-ARITHMETIC PRIMARY WHERE ARITHMETIC EXPECTED
156	NON-LOGICAL OPERAND WHERE LOGICAL EXPECTED
157	RELATIONAL OPERAND HAS LOGICAL TYPE
158	CHARACTER VS. ARITHMETIC RELATION
159	ILLEGAL RELATION FOR COMPLEX OPERANDS
160	OPERAND OF .NOT. NOT LOGICAL
161	IMPROPER SUBSTRING DESIGNATER
162	COMPLEX POWER
163	COMPLEX BASE TO NON-INTEGER POWER
164	STRING EXPRESSION IN PARENTHESES
165	PARTIAL-WORD EXCEEDS 15 BITS
166	IMPROPER TYPE FOR PARTIAL-WORD DESIGNATER
167	COMPLEX INDEX EXPRESSION
168	COMPLEX SUBSCRIPT
169	RECURSIVE STATEMENT FUNCTION
170	SUBROUTINE MISSING ARGUMENTS
171	FUNCTION MISSING ARGUMENTS
172	REDEFINITION OF USED INTRINSIC
173	MISSING SUBSCRIPT
174	ILLEGAL ARGUMENT FOR INTRINSIC
176	TOO FEW ARGUMENTS
177	TOO MANY ARGUMENTS
178	VALUE VS. REFERENCE ARGUMENT
179	CHARACTER ARGUMENT BY VALUE

Table D-1. FORTRAN/3000 Warning and Error Messages

180	NO LIMIT PARAMETER
181	TERMINAL LABEL PRECEDES DO STATEMENT
182	IMPROPERLY NESTED DO STATEMENTS
183	INTEGER SIMPLE VARIABLE EXPECTED
184	IMPROPER TERMINAL STATEMENT
185	UNDECLARED ARRAY NAME ?
186	LEFT-HAND IS FUNCTION OR SUBROUTINE
188	RIGHT AND LEFT-HAND TYPES INCOMPATIBLE
189	DUMMY HAS TYPE CHARACTER
191	CHARACTER STATEMENT FUNCTION
194	UNABLE TO CLASSIFY GOTO
196	IMPROPER ASSIGN
197	EXPECTED LOGICAL EXPRESSION
198	IMPROPER LOGICAL CLAUSE
199	IMPROPER DEPENDENT STATEMENT
200	ALTERNATE RETURN IN NON-SUBROUTINE
201	LABEL ARGUMENTS OUT OF POSITION
202	SYMBOL NOT SUBROUTINE NAME
203	EXPRESSION IN INPUT LIST
204	IMPROPER I/O LIST ITEM
205	EXPECTED I/O LIST
206	IMPROPER UNIT REFERENCE
207	EXPECTED CHARACTER VARIABLE
208	EXPECTED FORMAT REFERENCE
209	EXPECTED ACTION LABEL
210	DUPLICATE ACTION LABEL
211	TOO MANY TRACE SYMBOLS
212	DATA SPACE OVERFLOW
213	CODE SPACE OVERFLOW
:EOJ	

Table D-2. Compiler Error Message Action

Error Code	Compiler Action
0	The compiler appended the cause of termination as part of the message. STACK OVERFLOW, or SYMBOL TABLE OVERFLOW can cause termination.
7	The compiler found a character it did not expect and ignored it.
11	The compiler flushed the statement.
12	The compiler inserted the period.
4	The compiler inserted the comma.
5	The compiler ignored the comma.
16	A statement could have logically ended at some point prior to the physical end of the statement and the compiler could not logically phrase the next symbol as a continuation of the statement. The compiler flushed the remainder of the statement.
38	Too many continuation lines appear for one statement. The compiler flushed the execess continuation lines.
41	The compiler did not find a symbolic name necessary for proper statement form. The compiler flushed the statement.
43	The compiler discovered an improper statement label and ignored it.
51-52	The compiler expected a right or left parentheses and flushed the statement.
53	The compiler ignored the right parentheses ")".
55	The compiler expected a right bracket "]" and flushed the statement.
56	The compiler ignored the right bracked "]".
58	The compiler expected an assignment operator (=) and flushed the statement.
59	The compiler exprected a forward slash (/) and flushed the statement.
60	The compiler expected a string delimiter (') or (") but did not find one. The compiler flushed the statement.
61	The compiler expected a back slash (\) and flushed the statement.

Table D-2. Compiler Error Message Action

Error Code	Compiler Action
63	The compiler found a period (.) extraneous to the statement syntax requirements and ignored it.
64	The compiler found a "%" extraneous in the context used and ignored it.
65	The compiler found a dollar sign ($) which makes no sense in the context used and ignored it.
67	The compiler discovered a symbolic name exceeding 255 characters and flushed the statement containing the name.
68	The compiler discovered a number exceeding 255 characters and flushed the statement containing the number.
69	The compiler discovered a string literal exceeding 255 characters and flushed the statement containing the string.
70	The compiler discovered a Hollerith literal exceeding 255 characters and flushed the statement containing the literal.
71	The compiler discovered a Hollerith literal shorter than that indicated by the length element of the literal and flushed the statement containing the literal.
72	The compiler discovered a nonoctal digit in an octal number and ignored the octal number.
73	The compiler expected a field width to be indicated and flushed the statement when no field width was found.
74	The compiler expected a field value to be specified and flushed the statement when no field value was found.
75	A packed number overflowed and the compiler used whatever arbirary value was contained in the data space.
78	Integer overflow occurred and the compiler used whatever arbitrary value was contained in the data space.
80	A floating-point literal caused under- or over-flow and the compiler used whatever arbitrary value was contained in the data space.
76	This integer value cannot be equal to zero so the compiler changed it to 1.

Table D-2. Compiler Error Message Action

Error Code	Compiler Action
77	An integer value is incorrect according to the statement context, so the compiler changed it to 1.
79	An expected exponent value was incorrect or not supplied and the compiler flushed the statement.
81	The compiler discovered an improper complex literal and flushed the statement containing it.
84	The compiler expected an END line at this point and supplied one.
85	The compiler did not expect a continuation line to be used and flushed the entire statement up to (but not including) the next non-continuation line (start of next statement).
87	The compiler expected a reserved symbol such as .OR. or .TRUE. and did not recognize it. The compiler might have flushed the statement.
88	The compiler expected a FORTRAN/3000 keyword such as INTEGER or REAL but flushed the statement when the keyword was not recognized.
89	A statement possibly did not begin with a letter or its missing a right parentheses ")" causing the compiler to flush the statement.
91	The compiler has discovered a declaration statement following an executable statement or an exectable statement in a BLOCK-DATA subprogram, etc., and flushed the statement.
92	The compiler has discovered a dummy name not unique to the program unit in which the name appears. The compiler continues to scan.
93	The compiler has discovered an improper dummy argument and flushed the statement containing the argument.
94	A function or subroutine contains more than 54 arguments, or a statement function has too many arguments. The number of arguments allowed depends upon the argument type and complexity of the defining expression. The compiler continues to scan.
95	The compiler found too many alternate return points specified in a subprogram. The compiler containues to scan.
96	The compiler has discovered an incorrect type construct and flushed the statement.

Table D-2. Compiler Error Message Action

Error Code	Compiler Action
97	The compiler has discovered an incorrect type based on the first letter of the symbol and flushed the statement containing that symbol.
99	The compiler has found a subroutine subprogram with a type implicitly or explicitty declared but continued to scan.
100	The compiler has found a symbolic name whose type was declared in an inconsistant manner for its usage. The compiler continued to scan.
102	The compiler has discovered a dynamic bound defined in a DIMENSION statement but continued scanning.
103	The compiler found a procedure name in a DIMENSION statement but continued scanning.
104	The compiler found an array whose dimensions were defined more than once in the same program unit but continued scanning.
105	The compiler expected an array bound and flushed the statement.
106	The compiler discovered an array defined with more than 32767 elements and continued scanning.
107	The compiler discovered an array with more than 255 bounds and continued scanning.
108	The compiler discovered a dynamically-defined data space mentioned in a COMMON statement and continued scanning.
109	The compiler discovered a dummy name used in a COMMON statement and continued scanning.
110	The compiler discovered an item appearing in a COMMON statement twice and continued scanning.
111	The compiler discovered a name used as a COMMON block name and as the name of a procedure. The compiler continued scanning.
112	The compiler discovered a procedure name appearing in a COMMON statement and continued scanning.
113	The compiler discovered a character function defined with a dynamic length (using a variable length specification) and continued scanning.

Table D-2. Compiler Error Message Action

Error Code	Compiler Action
114	The compiler discovered a simple variable or array name used in an EXTERNAL statement. The compiler continued scanning.
116	The compiler discovered a dynamic structure (defined with variable memory allocation) used in an EQUIVALENCE statement. The compiler continued scanning.
117	The compiler discovered a dummy name in an EQUIVALENCE statement and continued scanning.
118	The compiler discovered a procedure name in an EQUIVALENCE statement and continued scanning.
120	The compiler discovered a dynamic structure in a DATA statement and continued scanning.
121	The compiler discovered a dummy name in a DATA statement and continued scanning.
122	The compiler discovered a procedure name in a DATA statement and continued scanning.
123	The compiler discovered a data item in common in a DATA statement and continued scanning.
124	The compiler expected an initial value for a data element and flushed the statement when none was found.
125	The compiler discovered an initial value not coinciding with the defined data element type. The compiler continued scanning.
126	The compiler discovered a minus or plus sign without a constant following it. The compiler continued scanning.
127	The compiler discovered an array element with the number of subscript expressions not equal to the number of defined bounds. The compiler continued scanning.
128	The compiler discovered an array that occupies more than 32767 words of memory. The compiler continued scanning.
129	The compiler discovered an array element subscript that indicates a memory location outside the defined bounds of the array. The compiler continued scanning.

Table D-2. Compiler Error Message Action

Error Code	Compiler Action
131	The compiler was unable to address all of the local variables in a program unit. The compiler bypassed the rest of the program unit and continued with the next program unit.
132	The compiler discovere a dynamic bound not represented by a dummy integer. The compiler continued scanning.
134	The compiler discovered a data block larger than allowed. The compiler continued scanning.
135	The compiler discovered a common block larger than allowed. The compiler continued acanning.
136	The compiler discovered a common block that has attempted to extend the common data space from the beginning instead of from the end. The compiler continued scanning.
137	The compiler discovered a group of EQUIVALENCE statement which equivalence too large a block of data. The compiler continued scanning.
140	The compiler discovered a data value aligned on a half-word (byte) boundary instead of a full-word boundary. The compiler continued scanning.
141	The compiler discovered an element defined in a data block used in an EQUIVALENCE statement that is also declared in a common block. The compiler continued scanning.
142	The compiler discovered two common blocks whose elements are equated through an EQUIVALENCE statement. The compiler continued scanning.
143	The compiler discovered array elements in an EQUIVALENCE statement which cause other elements of the arrays to equate improperly; or two elements are equated that require unique data space (e.g., label values).
144	The compiler discovered a simple variable with a subscript and continued scanning.
145	The compiler expected a statement label and flushed the statement when no label was found.
147	The compiler discovered a duplicate statement label and ignored the label.

Table D-2. Compiler Error Message Action

Error Code	Compiler Action
148	The compiler found a label referenced in a statement but never found a statement prefixed by that label. The compiler ignored the referenced label.
149	The compiler discovered a statement label reference to a non-FORMAT statement when a FORMAT statement was expected. The compiler continued scanning.
150	The compiler discovered a statement label reference to a non-executable statement. The compiler continued scanning.
153	The compiler discovered a subroutine name used as a primary and flushed the statement.
154	The compiler expected to find an arithmetic primary and flushed the statement when no primary was found.
155	The compiler found a nonarithmetic primary when it expected an arithmetic primary and flushed the statement.
156	The compiler expected to find a logical operand and flushed the statement when a nonlogical operand was found.
157	The compiler discovered an operand in a relation of type logical (instead of arithmetic) and flushed the statement.
158	The compiler expected an arithmetic relation and flushed the statement when a character relation was found.
159	The compiler discovered an illegal relational operator between two complex values and continued scanning.
160	The compiler discovered a nonlogical operand following a .NOT. operator and dlushed the statement.
161	The compiler discovered a substring designator in improper form and flushed the statement.
162	The compiler discovered a number raised to a complex power and continued scanning.
163	The compiler found a complex number raised to a noninteger power and continued scanning.
164	The compiler discovered a string expression in parentheses when it expected an arithmetic expression and flushed the statement.

Table D-2. Compiler Error Message Action

Error Code	Compiler Action
165	The compiler discovered a partial-word designator that specifies more than 15 bits and continued scanning.
166	The compiler discovered a partial-word designator of improper type and continued scanning.
167	The compiler discovered an index expression of type complex and continued scanning.
168	The compiler discovered a subscript value of type complex and continued scanning.
169	The compiler discovered a recursively defined statement function and flushed the statement.
170	The compiler discovered a subroutine with the improper number of arguments and continued scanning.
171	The compiler discovered a function with the improper number of arguments and continued scanning.
172	The compiler discovered an intrinsic that had been redefined after being called and flushed the statement.
173	The compiler discovered a missing subscript for an array name and flushed the statement.
174	The compiler discovered an illegal argument for an intrinsic and flushed the statement.
176	The compiler discovered a procedure with too few arguments and continued scanning.
177	The compiler discovered a procedure with too many arguments and continued scanning.
178	The compiler discovered an argument passed by value when it expected an argument passed by reference. The compiler continued scanning.
179	The compiler discovered a character argument being passed by value and continued scanning.
180	The compiler discovered a missing limit parameter in a DO statement or implied-DO in an I/O statement and continued scanning.

Table D-2. Compiler Error Message Action

Error Code	Compiler Action
181	The compiler discovered a DO-loop terminal label preceding the DO-loop DO statement and continued scanning.
182	The compiler discovered improperly nested DO statements and continued scanning.
183	The compiler expected an integer simple variable but did not find one. The compiler continued scanning.
184	The compiler discovered an improper terminal statement in a DO[loop and continued scanning.
185	The compiler discovered a symbolic name used as an array but not defined as such. The compiler flushed the statement.
186	The compiler discovered a function or subroutine on the left side of an assignment operator and flushed the statement.
188	The compiler discovered imcompatible types in the left and right sides of an assignment statement and continued scanning.
189	The compiler discovered a dummy parameter of type character and continued scanning.
191	The compiler discovered a character-type statement function and flushed the statement.
194	The compiler was unable to classify a GO TO statement as one of the three allowable types and dlushed the statement.
196	The compiler discovered an improper ASSIGN statement and flushed the statement.
197	The compiler expected a logical expression but continued scanning when none was found.
198	The compiler discovered an improper logical clause and flushed the statement.
199	The compiler discovered an improper dependent statement in an IF statement and continued scanning.
200	The compiler discovered an alternate RETURN statement in a nonsubroutine and flushed the statement.
201	The compiler discovered label arguments appearing elsewhere than at the end of the argument list and flushed the statement.

Table D-2. Compiler Error Message Action

Error Code	Compiler Action
202	The compiler expected a symbol to be a subroutine name and flushed the statement when none was found.
203	The compiler discovered an expression in an I/O input list and continued scanning.
204	The compiler discovered an improper I/O list item in an I/O statement and flushed the statement.
205	The compiler expected an I/O list and flushed the statement when none was found.
206	The compiler discovered an improper unit reference in an I/O statement and fxshed the statement.
207	The compiler expected a character variable and flushed the statement when none was found.
208	The compiler expected a format reference in an I/O statement and flushed the statement when none was found.
209	The compiler expected an action label in an I/O statement and flushed the statement when none was found.
210	The compiler discovered a duplicate action label and continued scanning.
211	The TRACE/3000 symbol table overflowed but the compiler continued to compile the program unit.
212	The program unit data space overflowed but the compiler contined to compile the program unit.
213	The program code space overflowed but the compiler continued to compile the program unit.

Table D-3. Compiler Warning Messages

Warning Code	Compiler Action
1	The compiler ignored the label field.
2	The compiler ignored the label field.
3	The compiler truncated the symbolic name to 15 characters.
4	The compiler assumed the comma was present and continued compilation.
5	The compiler ignored the comma and continued compilation.
6	The compiler expected a colon ":" and flushed the source record when none was found.
7	The compiler skipped to the next comma and ignored the statement between the extra character and the comman.
8	FORMAT warning — the compiler continued scanning.
9	FORMAT warning — the compiler continued scanning.
10	FORMAT warning — the compiler continued scanning.
11	FORMAT warning — the compiler continued scanning.
12	FORMAT warning — the compiler continued scanning.
13	FORMAT warning — the compiler continued scanning.
14	FORMAT warning — the compiler continued scanning.
15	FORMAT warning — the compiler continued scanning.
16	A statement could have logically ended at some point before the physical end of the statement and the compiler cannot logically parse the next symbol as a continuation of the statement. The compiler ignored the remainder of the record.
17	The compiler used the last seen segment name.
18	The compiler took no action and continued scanning.
19	The compiler took no action and continued scanning.
20	The compiler took no action and continued scanning.
22	The compiler ignores the extra values.

Table D-3. Compiler Warning Messages

Warning Code	Compiler Action
23	The compiler discovered a variable name in a DATA statement in a Block Data subprogram which is also not in a common block. The compiler continues scanning.
24	The compiler discovered a Block Data subprogram that did not specify initial values for a labeled common block and continued scanning.
25	The compiler truncated an initial value that was too large for its type.
26	The compiler takes no action and continues scanning.
27	The compiler discovered actual arguments inconsistant with the dummy arguments and continued scanning.
28	The compiler discovered actual arguments inconsistant with dummy argument and continued scanning.
29	The compiler discovered actual arguments inconsistant with dummy arguments and continued scanning.
30	The compiler discoverd actual arguments inconsistant with dummy arguments and continued scanning.
31	The compiler discovered a subprogram name not mentioned in an EXTERNAL statement and continued scanning.
32	The compiler discovered a recursive call in the actual argument list of an external procedure call and informed the user that only the actual arguments of the recursive call not the inital call, are checked.
33	The compiler converted an intrinsic name to a simple variable.
34	The compiler discovered a statement that cannot be locally addressed.
36	The compiler inserted a RETURN or STOP statement because an unlabeled END line was not preceded by an unconditional transfer of control.
38	The compiler discovered too many continuation lines for one statement and ignored the excess lines.
39	The compiler expected a continuation line and did not find one.

Table D-3. Compiler Warning Messages

Warning Code	Compiler Action
40	The compiler discovered a missing keyword, unrecognized keyword or incomplete parmeter. The compiler skipped past the next comma in the statement and continued scanning.
41	The compiler expected a symbolic name and continued scanning.
42	The compiler recognized a construct in the program unit that appeared too late to be honored and ignored the construct.
43	The compiler discovered a statement label equal to zero or greater than 99999 and ignored it.
44	The compiler cannot find the symbol mentioned in a TRACE record.
45	The compiler discovered an intrinsic name in a TRACE record. The intrinsic will not be traced.
77	The compiler discovered an integer equal to zero (divided into another number) or an integer too large for its intended usage.